```go
/* controller/components/components.go */

// Package components defines the consolidated Entity-Component-System (ECS) components
// for the CPRA monitoring application. This design follows the principles of data-oriented
// design to maximize performance and minimize memory usage, as required for handling
// over one million concurrent monitors.
//
// By consolidating state, configuration, and jobs into a few coarse-grained components,
// we dramatically reduce the number of archetypes in the ECS world. This leads to:
//   - Improved cache locality and iteration speed.
//   - Reduced memory fragmentation.
//   - Simplified system logic by avoiding complex component additions/removals for state transitions.
//
// State management is handled via a bitfield in the MonitorState component, allowing for
// efficient, atomic updates to an entity's status without changing its archetype.
package components

import (
	"cpra/internal/jobs"
	"cpra/internal/loader/schema"
	"errors"
	"strings"
	"sync/atomic"
	"time"
)

// MonitorState consolidates all monitor state into a single component.
// This approach dramatically reduces archetype fragmentation and improves cache locality.
type MonitorState struct {
	// Entity identification
	Name string

	// State flags (bitfield for efficiency) - replaces multiple tag components
	Flags uint32

	// Timing data
	LastCheckTime   time.Time
	LastSuccessTime time.Time
	NextCheckTime   time.Time

	// Error tracking
	ConsecutiveFailures int
	LastError           error

	// Pending action data
	PendingCode string
}

// State flag constants - replaces separate components like PulseNeeded, PulsePending, etc.
const (
	StateDisabled           uint32 = 1 << 0
	StatePulseNeeded        uint32 = 1 << 1
	StatePulsePending       uint32 = 1 << 2
	StatePulseFirstCheck    uint32 = 1 << 3
	StateInterventionNeeded uint32 = 1 << 4
	StateInterventionPending uint32 = 1 << 5
	StateCodeNeeded         uint32 = 1 << 6
	StateCodePending        uint32 = 1 << 7
	// Room for more states without adding components
)

// Efficient state management methods using atomic operations
func (m *MonitorState) IsDisabled() bool { return atomic.LoadUint32(&m.Flags)&StateDisabled != 0 }
func (m *MonitorState) IsPulseNeeded() bool {
	return atomic.LoadUint32(&m.Flags)&StatePulseNeeded != 0
}
func (m *MonitorState) IsPulsePending() bool {
	return atomic.LoadUint32(&m.Flags)&StatePulsePending != 0
}
func (m *MonitorState) IsPulseFirstCheck() bool {
	return atomic.LoadUint32(&m.Flags)&StatePulseFirstCheck != 0
}
func (m *MonitorState) IsInterventionNeeded() bool {
	return atomic.LoadUint32(&m.Flags)&StateInterventionNeeded != 0
}
func (m *MonitorState) IsInterventionPending() bool {
	return atomic.LoadUint32(&m.Flags)&StateInterventionPending != 0
}
```

```go
func (m *MonitorState) IsCodeNeeded() bool {
	return atomic.LoadUint32(&m.Flags)&StateCodeNeeded != 0
}
func (m *MonitorState) IsCodePending() bool {
	return atomic.LoadUint32(&m.Flags)&StateCodePending != 0
}

func (m *MonitorState) SetDisabled(disabled bool) {
	if disabled {
		atomic.OrUint32(&m.Flags, StateDisabled)
	} else {
		atomic.AndUint32(&m.Flags, ^StateDisabled)
	}
}

func (m *MonitorState) SetPulseNeeded(needed bool) {
	if needed {
		atomic.OrUint32(&m.Flags, StatePulseNeeded)
	} else {
		atomic.AndUint32(&m.Flags, ^StatePulseNeeded)
	}
}

func (m *MonitorState) SetPulsePending(pending bool) {
	if pending {
		atomic.OrUint32(&m.Flags, StatePulsePending)
	} else {
		atomic.AndUint32(&m.Flags, ^StatePulsePending)
	}
}

func (m *MonitorState) SetPulseFirstCheck(firstCheck bool) {
	if firstCheck {
		atomic.OrUint32(&m.Flags, StatePulseFirstCheck)
	} else {
		atomic.AndUint32(&m.Flags, ^StatePulseFirstCheck)
	}
}

func (m *MonitorState) SetInterventionNeeded(needed bool) {
	if needed {
		atomic.OrUint32(&m.Flags, StateInterventionNeeded)
	} else {
		atomic.AndUint32(&m.Flags, ^StateInterventionNeeded)
	}
}

func (m *MonitorState) SetInterventionPending(pending bool) {
	if pending {
		atomic.OrUint32(&m.Flags, StateInterventionPending)
	} else {
		atomic.AndUint32(&m.Flags, ^StateInterventionPending)
	}
}

func (m *MonitorState) SetCodeNeeded(needed bool) {
	if needed {
		atomic.OrUint32(&m.Flags, StateCodeNeeded)
	} else {
		atomic.AndUint32(&m.Flags, ^StateCodeNeeded)
	}
}

func (m *MonitorState) SetCodePending(pending bool) {
	if pending {
		atomic.OrUint32(&m.Flags, StateCodePending)
	} else {
		atomic.AndUint32(&m.Flags, ^StateCodePending)
	}
}

// PulseConfig consolidates pulse configuration
type PulseConfig struct {
	Type        string
	Timeout     time.Duration
	Interval    time.Duration
	Retries     int
	MaxFailures int
	Config      schema.PulseConfig
}
```

```go
func (c *PulseConfig) Copy() *PulseConfig {
	if c == nil {
		return nil
	}
	cpy := &PulseConfig{
		Type:        strings.Clone(c.Type),
		Timeout:     c.Timeout,
		Interval:    c.Interval,
		Retries:     c.Retries,
		MaxFailures: c.MaxFailures,
	}

	if c.Config != nil {
		cpy.Config = c.Config.Copy()
	}
	return cpy
}

// InterventionConfig consolidates intervention configuration
type InterventionConfig struct {
	Action      string
	MaxFailures int
	Target      schema.InterventionTarget
}

func (c *InterventionConfig) Copy() *InterventionConfig {
	if c == nil {
		return nil
	}
	cpy := &InterventionConfig{
		Action:      strings.Clone(c.Action),
		MaxFailures: c.MaxFailures,
	}

	if c.Target != nil {
		cpy.Target = c.Target.Copy()
	}
	return cpy
}

// CodeConfig consolidates all code configurations instead of separate color components.
// This single component replaces RedCodeConfig, GreenCodeConfig, CyanCodeConfig, etc.
type CodeConfig struct {
	// Color-specific configurations stored as map instead of separate components
	Configs map[string]*ColorCodeConfig
}

type ColorCodeConfig struct {
	Dispatch    bool
	MaxFailures int
	Notify      string
	Config      schema.CodeNotification
}

func (c *ColorCodeConfig) Copy() *ColorCodeConfig {
	if c == nil {
		return nil
	}
	cpy := &ColorCodeConfig{
		Dispatch:    c.Dispatch,
		MaxFailures: c.MaxFailures,
		Notify:      strings.Clone(c.Notify),
	}
	if c.Config != nil {
		cpy.Config = c.Config.Copy()
	}
	return cpy
}

func (c *CodeConfig) Copy() *CodeConfig {
	if c == nil {
		return nil
	}
	cpy := &CodeConfig{
		Configs: make(map[string]*ColorCodeConfig),
	}
	for color, config := range c.Configs {
		cpy.Configs[color] = config.Copy()
	}
```

```go
        return cpy
}

// CodeStatus consolidates all code status instead of separate color status components
type CodeStatus struct {
        // Color-specific status stored as map
        Status map[string]*ColorCodeStatus
}

type ColorCodeStatus struct {
        LastStatus          string
        ConsecutiveFailures int
        LastAlertTime       time.Time
        LastSuccessTime     time.Time
        LastError           error
}

func (s *ColorCodeStatus) SetSuccess(t time.Time) {
        s.LastStatus = "success"
        s.LastError = nil
        s.ConsecutiveFailures = 0
        s.LastSuccessTime = t
        s.LastAlertTime = t
}

func (s *ColorCodeStatus) SetFailure(err error) {
        s.LastStatus = "failed"
        s.LastError = err
        s.ConsecutiveFailures++
}

func (s *ColorCodeStatus) Copy() *ColorCodeStatus {
        if s == nil {
                return nil
        }
        cpy := &ColorCodeStatus{
                LastStatus:          strings.Clone(s.LastStatus),
                ConsecutiveFailures: s.ConsecutiveFailures,
                LastAlertTime:       s.LastAlertTime,
                LastSuccessTime:     s.LastSuccessTime,
        }
        if s.LastError != nil {
                cpy.LastError = errors.New(s.LastError.Error())
        }
        return cpy
}

func (c *CodeStatus) Copy() *CodeStatus {
        if c == nil {
                return nil
        }
        cpy := &CodeStatus{
                Status: make(map[string]*ColorCodeStatus),
        }
        for color, status := range c.Status {
                cpy.Status[color] = status.Copy()
        }
        return cpy
}

// JobStorage consolidates all job storage instead of separate job components.
// This single component replaces PulseJob, InterventionJob, CodeJob, etc.
type JobStorage struct {
        PulseJob        jobs.Job
        InterventionJob jobs.Job
        CodeJobs        map[string]jobs.Job // Jobs for each code color
}

func (j *JobStorage) Copy() *JobStorage {
        if j == nil {
                return nil
        }
        cpy := &JobStorage{
                CodeJobs: make(map[string]jobs.Job),
        }
        if j.PulseJob != nil {
                cpy.PulseJob = j.PulseJob.Copy()
        }
        if j.InterventionJob != nil {
                cpy.InterventionJob = j.InterventionJob.Copy()
```

```go
	}
	for color, job := range j.CodeJobs {
		if job != nil {
			cpy.CodeJobs[color] = job.Copy()
		}
	}
	return cpy
}

// Result components are used to convey job completion information back to the ECS.
// They are added to entities by the result handling logic and removed by the corresponding result
system.

type PulseResult struct {
	Result jobs.Result
}

type InterventionResult struct {
	Result jobs.Result
}

type CodeResult struct {
	Result jobs.Result
}


/* controller/entities/mapper.go */

package entities

import (
	"cpra/internal/controller/components"
	"cpra/internal/jobs"
	"cpra/internal/loader/schema"
	"fmt"
	"strings"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// EntityManager uses the new consolidated component design.
// This dramatically reduces the number of archetypes and improves performance.
type EntityManager struct {
	// Core consolidated components - only a few archetypes instead of dozens.
	MonitorState       *ecs.Map1[components.MonitorState]
	PulseConfig        *ecs.Map1[components.PulseConfig]
	InterventionConfig *ecs.Map1[components.InterventionConfig]
	CodeConfig         *ecs.Map1[components.CodeConfig]
	CodeStatus         *ecs.Map1[components.CodeStatus]
	JobStorage         *ecs.Map1[components.JobStorage]
}

// NewEntityManager creates a new consolidated entity manager.
func NewEntityManager(world *ecs.World) *EntityManager {
	return &EntityManager{
		MonitorState:       ecs.NewMap1[components.MonitorState](world),
		PulseConfig:        ecs.NewMap1[components.PulseConfig](world),
		InterventionConfig: ecs.NewMap1[components.InterventionConfig](world),
		CodeConfig:         ecs.NewMap1[components.CodeConfig](world),
		CodeStatus:         ecs.NewMap1[components.CodeStatus](world),
		JobStorage:         ecs.NewMap1[components.JobStorage](world),
	}
}

// CreateEntityFromMonitor creates an entity using the consolidated design.
func (e *EntityManager) CreateEntityFromMonitor(
	monitor *schema.Monitor,
	world *ecs.World) error {

	// Validation
	if world == nil {
		return fmt.Errorf("world cannot be nil")
	}
	if e == nil {
		return fmt.Errorf("EntityManager cannot be nil")
	}
	if monitor.Name == "" {
		fmt.Println(monitor, "name cannot be empty")
		return fmt.Errorf("monitor name cannot be empty")
```

```go
	}

	entity := world.NewEntity()
	if !world.Alive(entity) {
		return fmt.Errorf("failed to create valid entity")
	}

	// Create consolidated MonitorState component
	monitorState := &components.MonitorState{
		Name:            strings.Clone(monitor.Name),
		LastCheckTime:   time.Now(),
		LastSuccessTime: time.Now(),
		NextCheckTime:   time.Now(),
	}

	// Set initial state flags
	if monitor.Enabled {
		monitorState.SetPulseFirstCheck(true) // Equivalent to adding PulseFirstCheck
component
	} else {
		monitorState.SetDisabled(true) // Equivalent to adding DisabledMonitor component
	}

	e.MonitorState.Add(entity, monitorState)

	// Add pulse configuration
	pulseConfig := &components.PulseConfig{
		Type:        strings.Clone(monitor.Pulse.Type),
		MaxFailures: monitor.Pulse.MaxFailures,
		Timeout:     monitor.Pulse.Timeout,
		Interval:    monitor.Pulse.Interval,
		Config:      monitor.Pulse.Config.Copy(),
	}
	e.PulseConfig.Add(entity, pulseConfig)

	// Create consolidated job storage
	jobStorage := &components.JobStorage{
		CodeJobs: make(map[string]jobs.Job),
	}

	// Add pulse job
	pulseJob, err := jobs.CreatePulseJob(monitor.Pulse, entity)
	if err != nil {
		return err
	}
	jobStorage.PulseJob = pulseJob

	// Add intervention if configured
	if monitor.Intervention.Action != "" {
		maxFailures := 1
		if monitor.Intervention.MaxFailures > 0 {
			maxFailures = monitor.Intervention.MaxFailures
		}

		interventionConfig := &components.InterventionConfig{
			Action:      strings.Clone(monitor.Intervention.Action),
			Target:      monitor.Intervention.Target.Copy(),
			MaxFailures: maxFailures,
		}
		e.InterventionConfig.Add(entity, interventionConfig)

		// Add intervention job
		interventionJob, err := jobs.CreateInterventionJob(monitor.Intervention, entity)
		if err != nil {
			return err
		}
		jobStorage.InterventionJob = interventionJob
	}

	// Add consolidated code configuration instead of separate color components
	if len(monitor.Codes) > 0 {
		codeConfig := &components.CodeConfig{
			Configs: make(map[string]*components.ColorCodeConfig),
		}
		codeStatus := &components.CodeStatus{
			Status: make(map[string]*components.ColorCodeStatus),
		}

		for color, config := range monitor.Codes {
			// Single consolidated entry instead of separate components
```

```
			codeConfig.Configs[color] = &components.ColorCodeConfig{
				Dispatch: config.Dispatch,
				Notify:   strings.Clone(config.Notify),
				Config:   config.Config.Copy(),
			}

			codeStatus.Status[color] = &components.ColorCodeStatus{
				LastAlertTime: time.Now(),
			}

			// Add code job to consolidated storage
			codeJob, err := jobs.CreateCodeJob(strings.Clone(monitor.Name), config,
entity, color)
			if err != nil {
				return err
			}
			jobStorage.CodeJobs[color] = codeJob
		}

		e.CodeConfig.Add(entity, codeConfig)
		e.CodeStatus.Add(entity, codeStatus)
	}

	e.JobStorage.Add(entity, jobStorage)

	return nil
}

// EnableMonitor enables a monitor using consolidated state flags
func (e *EntityManager) EnableMonitor(entity ecs.Entity) {
	if state := e.MonitorState.Get(entity); state != nil {
		state.SetDisabled(false)
		state.SetPulseFirstCheck(true)
	}
}

// DisableMonitor disables a monitor using consolidated state flags
func (e *EntityManager) DisableMonitor(entity ecs.Entity) {
	if state := e.MonitorState.Get(entity); state != nil {
		state.SetDisabled(true)
		state.SetPulsePending(false)
		state.SetInterventionPending(false)
		state.SetCodePending(false)
	}
}

// GetMonitorState provides easy access to consolidated state
func (e *EntityManager) GetMonitorState(entity ecs.Entity) *components.MonitorState {
	return e.MonitorState.Get(entity)
}


/* controller/logger.go */

package controller

import (
	"context"
	"fmt"
	"log"
	"os"
	"strings"
	"time"
)

// LogLevel represents different logging levels
type LogLevel int

const (
	LogLevelDebug LogLevel = iota
	LogLevelInfo
	LogLevelWarn
	LogLevelError
	LogLevelFatal
)

var logLevelNames = map[LogLevel]string{
	LogLevelDebug: "DEBUG",
	LogLevelInfo:  "INFO",
	LogLevelWarn:  "WARN",
```

```go
	LogLevelError: "ERROR",
	LogLevelFatal: "FATAL",
}

var logLevelColors = map[LogLevel]string{
	LogLevelDebug: "\033[36m", // Cyan
	LogLevelInfo:  "\033[32m", // Green
	LogLevelWarn:  "\033[33m", // Yellow
	LogLevelError: "\033[31m", // Red
	LogLevelFatal: "\033[35m", // Magenta
}

const colorReset = "\033[0m"

// Logger provides structured logging with levels and context
type Logger struct {
	level       LogLevel
	component   string
	enableColor bool
	debugMode   bool
	prodMode    bool
	file        *os.File
	timezone    *time.Location
	tracer      *Tracer
}

// NewLogger creates a new logger instance
func NewLogger(component string, debugMode bool) *Logger {
	level := LogLevelInfo
	if debugMode {
		level = LogLevelDebug
	}

	// Check environment for production mode
	prodMode := strings.ToLower(os.Getenv("CPRA_ENV")) == "production"
	if prodMode {
		level = LogLevelWarn // More restrictive in production
	}

	// Enable colors for terminal output (disable in production)
	enableColor := !prodMode && isTerminal()

	// Get timezone from environment or use local timezone
	timezone := getTimezone()

	// Enable tracing in debug mode or if explicitly enabled
	enableTracing := debugMode || strings.ToLower(os.Getenv("CPRA_TRACING")) == "true"

	logger := &Logger{
		level:       level,
		component:   component,
		enableColor: enableColor,
		debugMode:   debugMode,
		prodMode:    prodMode,
		timezone:    timezone,
	}

	// Setup file logging for production
	if prodMode {
		logger.setupFileLogging()
	}

	// Setup tracing if enabled
	if enableTracing {
		logger.tracer = NewTracer(component, true)
	}

	return logger
}

// getTimezone returns the timezone to use for logging
func getTimezone() *time.Location {
	// Check environment variable first
	if tz := os.Getenv("CPRA_TIMEZONE"); tz != "" {
		if loc, err := time.LoadLocation(tz); err == nil {
			return loc
		}
		log.Printf("Warning: Invalid timezone '%s', using local timezone", tz)
	}
```

```go
	// Use local timezone as default
	return time.Local
}

// isTerminal checks if we're running in a terminal
func isTerminal() bool {
	fileInfo, _ := os.Stdout.Stat()
	return (fileInfo.Mode() & os.ModeCharDevice) != 0
}

// setupFileLogging configures file output for production
func (l *Logger) setupFileLogging() {
	logFile := fmt.Sprintf("cpra-%s.log", time.Now().In(l.timezone).Format("2006-01-02"))
	file, err := os.OpenFile(logFile, os.O_CREATE|os.O_WRONLY|os.O_APPEND, 0644)
	if err != nil {
		log.Printf("Failed to open log file: %v", err)
		return
	}
	l.file = file
}

// Close closes the log file if open
func (l *Logger) Close() {
	if l.file != nil {
		l.file.Close()
	}
}

// formatMessage formats a log message with timestamp, level, and component
func (l *Logger) formatMessage(level LogLevel, msg string, args ...interface{}) string {
	// Use enhanced timestamp with timezone information - 12-hour format with AM/PM
	now := time.Now().In(l.timezone)
	timestamp := now.Format("2006-01-02 03:04:05.000 PM Z07:00") // 12-hour format with space and
AM/PM
	timezoneName := l.timezone.String()
	levelName := logLevelNames[level]

	formattedMsg := fmt.Sprintf(msg, args...)

	// Add tracing info if available
	traceInfo := ""
	if l.tracer != nil && l.tracer.enabled {
		stats := l.tracer.GetStats()
		if totalSpans, ok := stats["total_spans"].(int); ok && totalSpans > 0 {
			traceInfo = fmt.Sprintf(" [TRACE:spans=%d]", totalSpans)
		}
	}

	if l.enableColor {
		color := logLevelColors[level]
		return fmt.Sprintf("%s %s [%s%s%s] [%s]%s %s",
			timestamp, timezoneName, color, levelName, colorReset, l.component, traceInfo,
formattedMsg)
	}

	return fmt.Sprintf("%s %s [%s] [%s]%s %s",
		timestamp, timezoneName, levelName, l.component, traceInfo, formattedMsg)
}

// log writes a message at the specified level
func (l *Logger) log(level LogLevel, msg string, args ...interface{}) {
	if level < l.level {
		return
	}

	formatted := l.formatMessage(level, msg, args...)

	// Always output to stdout/stderr
	if level >= LogLevelError {
		fmt.Fprintf(os.Stderr, "%s\n", formatted)
	} else {
		fmt.Fprintf(os.Stdout, "%s\n", formatted)
	}

	// Also write to file in production
	if l.file != nil {
		fmt.Fprintf(l.file, "%s\n", formatted)
		l.file.Sync() // Ensure immediate write
	}
}
```

```go
// Debug logs a debug message (only in debug mode)
func (l *Logger) Debug(msg string, args ...interface{}) {
	l.log(LogLevelDebug, msg, args...)
}

// Info logs an info message
func (l *Logger) Info(msg string, args ...interface{}) {
	l.log(LogLevelInfo, msg, args...)
}

// Warn logs a warning message
func (l *Logger) Warn(msg string, args ...interface{}) {
	l.log(LogLevelWarn, msg, args...)
}

// Error logs an error message
func (l *Logger) Error(msg string, args ...interface{}) {
	l.log(LogLevelError, msg, args...)
}

// Fatal logs a fatal message and exits
func (l *Logger) Fatal(msg string, args ...interface{}) {
	l.log(LogLevelFatal, msg, args...)
	os.Exit(1)
}

// WithContext creates a new logger with additional context
func (l *Logger) WithContext(context string) *Logger {
	return &Logger{
		level:       l.level,
		component:   fmt.Sprintf("%s:%s", l.component, context),
		enableColor: l.enableColor,
		debugMode:   l.debugMode,
		prodMode:    l.prodMode,
		file:        l.file,
		timezone:    l.timezone,
		tracer:      l.tracer,
	}
}

// StartTrace begins a new trace span with the logger's tracer
func (l *Logger) StartTrace(ctx context.Context, operation string) (context.Context, *TraceSpan) {
	if l.tracer != nil {
		return l.tracer.StartSpan(ctx, operation)
	}
	return ctx, nil
}

// FinishTrace completes a trace span
func (l *Logger) FinishTrace(span *TraceSpan, err error) {
	if l.tracer != nil {
		l.tracer.FinishSpan(span, err)
	}
}

// AddTraceTag adds a tag to a trace span
func (l *Logger) AddTraceTag(span *TraceSpan, key, value string) {
	if l.tracer != nil {
		l.tracer.AddSpanTag(span, key, value)
	}
}

// AddTraceMetadata adds metadata to a trace span
func (l *Logger) AddTraceMetadata(span *TraceSpan, key string, value interface{}) {
	if l.tracer != nil {
		l.tracer.AddSpanMetadata(span, key, value)
	}
}

// SetTraceEntity sets the entity ID for a trace span
func (l *Logger) SetTraceEntity(span *TraceSpan, entityID uint64) {
	if l.tracer != nil {
		l.tracer.SetSpanEntity(span, entityID)
	}
}

// GetTracingStats returns tracing statistics
func (l *Logger) GetTracingStats() map[string]interface{} {
	if l.tracer != nil {
```

```go
		return l.tracer.GetStats()
	}
	return map[string]interface{}{"enabled": false}
}

// LogSystemPerformance logs system performance metrics
func (l *Logger) LogSystemPerformance(component string, duration time.Duration, entitiesProcessed int)
{
	if l.debugMode {
		rate := float64(entitiesProcessed) / duration.Seconds()
		l.Debug("Performance: %s processed %d entities in %v (%.1f/sec)",
			component, entitiesProcessed, duration, rate)
	}
}

// LogEntityOperation logs entity-level operations in debug mode
func (l *Logger) LogEntityOperation(operation string, entityID uint64, details string) {
	if l.debugMode {
		l.Debug("Entity[%d] %s: %s", entityID, operation, details)
	}
}

// LogWorkerPool logs worker pool statistics - debug only, completely silent otherwise
func (l *Logger) LogWorkerPool(poolName string, stats map[string]interface{}) {
	// Only log in debug mode, completely silent otherwise
}

// LogComponentState logs component state changes
func (l *Logger) LogComponentState(entityID uint32, component string, state string) {
	if l.debugMode {
		l.Debug("Entity[%d] %s -> %s", entityID, component, state)
	}
}

// LogChannelState logs channel buffer states
func (l *Logger) LogChannelState(channelName string, depth, capacity int) {
	if l.debugMode {
		utilization := float64(depth) / float64(capacity) * 100
		l.Debug("Channel[%s] depth: %d/%d (%.1f%% full)",
			channelName, depth, capacity, utilization)
	} else if depth == capacity {
		l.Warn("Channel[%s] is full (%d/%d)", channelName, depth, capacity)
	}
}

// LogJobExecution logs job execution details
func (l *Logger) LogJobExecution(jobType string, entityID uint64, duration time.Duration, success
bool) {
	if l.debugMode {
		status := "SUCCESS"
		if !success {
			status = "FAILED"
		}
		l.Debug("Job[%s] Entity[%d] %s in %v", jobType, entityID, status, duration)
	}
}

// Global logger instances for different components
var (
	SystemLogger     *Logger
	SchedulerLogger  *Logger
	DispatchLogger   *Logger
	ResultLogger     *Logger
	WorkerPoolLogger *Logger
	EntityLogger     *Logger
)

// InitializeLoggers sets up all component loggers
func InitializeLoggers(debugMode bool) {
	SystemLogger = NewLogger("SYSTEM", debugMode)
	SchedulerLogger = NewLogger("SCHEDULER", debugMode)
	DispatchLogger = NewLogger("DISPATCH", debugMode)
	ResultLogger = NewLogger("RESULT", debugMode)
	WorkerPoolLogger = NewLogger("WORKER", debugMode)
	EntityLogger = NewLogger("ENTITY", debugMode)
}

// CloseLoggers closes all logger files
func CloseLoggers() {
	loggers := []*Logger{
```

```go
		SystemLogger, SchedulerLogger, DispatchLogger,
		ResultLogger, WorkerPoolLogger, EntityLogger,
	}

	for _, logger := range loggers {
		if logger != nil {
			logger.Close()
		}
	}
}


/* controller/memory.go */

package controller

import (
	"log"
	"runtime"
	"runtime/debug"
	"time"
)

// MemoryManager handles memory optimization and monitoring
type MemoryManager struct {
	maxMemory     uint64
	gcInterval    time.Duration
	lastGC        time.Time
	memoryStats   runtime.MemStats
	alertThreshold float64 // Percentage of max memory before alert
}

func NewMemoryManager(maxMemoryGB uint64, gcIntervalSeconds int) *MemoryManager {
	return &MemoryManager{
		maxMemory:      maxMemoryGB << 30, // Convert GB to bytes
		gcInterval:     time.Duration(gcIntervalSeconds) * time.Second,
		alertThreshold: 0.8, // Alert at 80% memory usage
	}
}

// MonitorMemory checks current memory usage and triggers cleanup if needed
func (m *MemoryManager) MonitorMemory() {
	runtime.ReadMemStats(&m.memoryStats)

	currentUsage := m.memoryStats.Alloc
	usagePercent := float64(currentUsage) / float64(m.maxMemory)

	if usagePercent > m.alertThreshold {
		log.Printf("HIGH MEMORY USAGE: %.2f%% (%d MB / %d MB)",
			usagePercent*100,
			currentUsage>>20,
			m.maxMemory>>20)

		// Force garbage collection
		m.ForceGC()
	}

	// Periodic garbage collection
	if time.Since(m.lastGC) > m.gcInterval {
		runtime.GC()
		m.lastGC = time.Now()
	}
}

// ForceGC triggers immediate garbage collection with logging
func (m *MemoryManager) ForceGC() {
	before := m.memoryStats.Alloc
	runtime.GC()
	runtime.ReadMemStats(&m.memoryStats)
	after := m.memoryStats.Alloc

	freed := before - after
	log.Printf("Forced GC: freed %d MB (before: %d MB, after: %d MB)",
		freed>>20, before>>20, after>>20)

	m.lastGC = time.Now()
}

// GetMemoryStats returns current memory statistics
func (m *MemoryManager) GetMemoryStats() runtime.MemStats {
```

```
	runtime.ReadMemStats(&m.memoryStats)
	return m.memoryStats
}

// SetMemoryLimit configures runtime memory limits
func (m *MemoryManager) SetMemoryLimit() {
	debug.SetMemoryLimit(int64(m.maxMemory))
	log.Printf("Memory limit set to: %d GB", m.maxMemory>>30)
}

// LogMemoryStats provides detailed memory information
func (m *MemoryManager) LogMemoryStats() {
	stats := m.GetMemoryStats()

	log.Printf("Memory Stats:")
	log.Printf("  Alloc: %d MB", stats.Alloc>>20)
	log.Printf("  TotalAlloc: %d MB", stats.TotalAlloc>>20)
	log.Printf("  Sys: %d MB", stats.Sys>>20)
	log.Printf("  NumGC: %d", stats.NumGC)
	log.Printf("  GCCPUFraction: %.4f", stats.GCCPUFraction)
}

/* controller/metrics.go */

package controller

import (
	"sync"
	"time"
)

// SystemMetrics holds performance metrics for a specific system
type SystemMetrics struct {
	SystemName          string
	TotalUpdates        int64
	TotalEntitiesProcessed int64
	TotalBatchesCreated    int64
	TotalDuration          time.Duration
	MaxUpdateDuration      time.Duration
	MinUpdateDuration      time.Duration
	LastUpdateTime         time.Time
	StartTime              time.Time
}

// MetricsAggregator collects and aggregates metrics from all systems
type MetricsAggregator struct {
	mu      sync.RWMutex
	systems map[string]*SystemMetrics
}

// NewMetricsAggregator creates a new metrics aggregator
func NewMetricsAggregator() *MetricsAggregator {
	return &MetricsAggregator{
		systems: make(map[string]*SystemMetrics),
	}
}

// RegisterSystem registers a new system for metrics collection
func (ma *MetricsAggregator) RegisterSystem(systemName string) {
	ma.mu.Lock()
	defer ma.mu.Unlock()

	ma.systems[systemName] = &SystemMetrics{
		SystemName:        systemName,
		StartTime:         time.Now(),
		MinUpdateDuration: time.Hour, // Initialize to high value
	}
}

// RecordSystemUpdate records a system update with performance metrics
func (ma *MetricsAggregator) RecordSystemUpdate(systemName string, duration time.Duration,
entitiesProcessed int64, batchesCreated int64) {
	ma.mu.Lock()
	defer ma.mu.Unlock()

	metrics, exists := ma.systems[systemName]
	if !exists {
		// Auto-register system if not found
		metrics = &SystemMetrics{
			SystemName:        systemName,
```

```
				StartTime:          time.Now(),
				MinUpdateDuration: time.Hour,
			}
			ma.systems[systemName] = metrics
		}

		metrics.TotalUpdates++
		metrics.TotalEntitiesProcessed += entitiesProcessed
		metrics.TotalBatchesCreated += batchesCreated
		metrics.TotalDuration += duration
		metrics.LastUpdateTime = time.Now()

		// Update min/max durations
		if duration > metrics.MaxUpdateDuration {
			metrics.MaxUpdateDuration = duration
		}
		if duration < metrics.MinUpdateDuration {
			metrics.MinUpdateDuration = duration
		}
}

// GetSystemMetrics returns metrics for a specific system
func (ma *MetricsAggregator) GetSystemMetrics(systemName string) (*SystemMetrics, bool) {
		ma.mu.RLock()
		defer ma.mu.RUnlock()

		metrics, exists := ma.systems[systemName]
		if !exists {
			return nil, false
		}

		// Return a copy to avoid race conditions
		copy := *metrics
		return ©, true
}

// GetAllMetrics returns metrics for all systems
func (ma *MetricsAggregator) GetAllMetrics() map[string]*SystemMetrics {
		ma.mu.RLock()
		defer ma.mu.RUnlock()

		result := make(map[string]*SystemMetrics)
		for name, metrics := range ma.systems {
			copy := *metrics
			result[name] = ©
		}

		return result
}

// GetAggregateMetrics returns aggregate metrics across all systems
func (ma *MetricsAggregator) GetAggregateMetrics() AggregateMetrics {
		ma.mu.RLock()
		defer ma.mu.RUnlock()

		var aggregate AggregateMetrics
		aggregate.StartTime = time.Now()

		for _, metrics := range ma.systems {
			aggregate.TotalUpdates += metrics.TotalUpdates
			aggregate.TotalEntitiesProcessed += metrics.TotalEntitiesProcessed
			aggregate.TotalBatchesCreated += metrics.TotalBatchesCreated
			aggregate.TotalDuration += metrics.TotalDuration
			aggregate.SystemCount++

			if metrics.StartTime.Before(aggregate.StartTime) {
				aggregate.StartTime = metrics.StartTime
			}
			if metrics.MaxUpdateDuration > aggregate.MaxUpdateDuration {
				aggregate.MaxUpdateDuration = metrics.MaxUpdateDuration
			}
			if aggregate.MinUpdateDuration == 0 || (metrics.MinUpdateDuration <
aggregate.MinUpdateDuration && metrics.MinUpdateDuration > 0) {
				aggregate.MinUpdateDuration = metrics.MinUpdateDuration
			}
		}

		// Calculate averages
		if aggregate.TotalUpdates > 0 {
			aggregate.AvgUpdateDuration = aggregate.TotalDuration /
```

```
time.Duration(aggregate.TotalUpdates)
                aggregate.AvgEntitiesPerUpdate = float64(aggregate.TotalEntitiesProcessed) /
float64(aggregate.TotalUpdates)
                aggregate.AvgBatchesPerUpdate = float64(aggregate.TotalBatchesCreated) /
float64(aggregate.TotalUpdates)
        }

        // Calculate throughput
        totalRuntime := time.Since(aggregate.StartTime)
        if totalRuntime > 0 {
                aggregate.EntitiesPerSecond = float64(aggregate.TotalEntitiesProcessed) /
totalRuntime.Seconds()
                aggregate.UpdatesPerSecond = float64(aggregate.TotalUpdates) / totalRuntime.Seconds()
        }

        return aggregate
}

// AggregateMetrics holds aggregate performance metrics across all systems
type AggregateMetrics struct {
        SystemCount            int
        TotalUpdates           int64
        TotalEntitiesProcessed int64
        TotalBatchesCreated    int64
        TotalDuration          time.Duration
        MaxUpdateDuration      time.Duration
        MinUpdateDuration      time.Duration
        AvgUpdateDuration      time.Duration
        AvgEntitiesPerUpdate   float64
        AvgBatchesPerUpdate    float64
        EntitiesPerSecond      float64
        UpdatesPerSecond       float64
        StartTime              time.Time
}

/* controller/optimized_controller.go */

package controller

import (
        "context"
        "cpra/internal/controller/systems"
        "cpra/internal/queue"
        "fmt"
        "log"
        "math"
        "os"
        "time"

        "cpra/internal/controller/entities"
        "cpra/internal/loader/streaming"
        "github.com/mlange-42/ark-tools/app"
        "github.com/mlange-42/ark/ecs"
)

// LoggerAdapter adapts the controller loggers to the systems interface.
type LoggerAdapter struct {
        logger interface {
                Info(format string, args ...interface{})
                Debug(format string, args ...interface{})
                Warn(format string, args ...interface{})
                Error(format string, args ...interface{})
                LogSystemPerformance(name string, duration time.Duration, count int)
        }
}

func (l *LoggerAdapter) Info(format string, args ...interface{})  { l.logger.Info(format, args...) }
func (l *LoggerAdapter) Debug(format string, args ...interface{}) { l.logger.Debug(format, args...) }
func (l *LoggerAdapter) Warn(format string, args ...interface{})  { l.logger.Warn(format, args...) }
func (l *LoggerAdapter) Error(format string, args ...interface{}) { l.logger.Error(format, args...) }
func (l *LoggerAdapter) LogSystemPerformance(name string, duration time.Duration, count int) {
        l.logger.LogSystemPerformance(name, duration, count)
}
func (l *LoggerAdapter) LogComponentState(entityID uint32, component string, action string) {
        l.logger.Debug("Entity[%d] component %s: %s", entityID, component, action)
}

// OptimizedController manages the ECS world and its systems using ark-tools.
type OptimizedController struct {
        app    *app.App
```

```go
	world  *ecs.World
	mapper *entities.EntityManager

	pulseQueue        queue.Queue
	interventionQueue queue.Queue
	codeQueue         queue.Queue

	pulsePool        *queue.DynamicWorkerPool
	interventionPool *queue.DynamicWorkerPool
	codePool         *queue.DynamicWorkerPool

	// ECS Systems
	pulseScheduleSystem      *systems.BatchPulseScheduleSystem
	pulseSystem              *systems.BatchPulseSystem
	pulseResultSystem        *systems.BatchPulseResultSystem
	interventionSystem       *systems.BatchInterventionSystem
	interventionResultSystem *systems.BatchInterventionResultSystem
	codeSystem               *systems.BatchCodeSystem
	codeResultSystem         *systems.BatchCodeResultSystem

	config  Config
	running bool
}

// Config holds all configuration for the controller.
type Config struct {
	StreamingConfig streaming.StreamingConfig
	QueueCapacity   uint64
	WorkerConfig    queue.WorkerPoolConfig
	BatchSize       int
	UpdateInterval  time.Duration
}

// DefaultConfig returns a default configuration.
func DefaultConfig() Config {
	return Config{
		StreamingConfig: streaming.DefaultStreamingConfig(),
		QueueCapacity:   65536, // Must be a power of 2
		WorkerConfig:    queue.DefaultWorkerPoolConfig(),
		BatchSize:       1000,
		UpdateInterval:  100 * time.Millisecond,
	}
}

// NewOptimizedController creates a new controller with the refactored systems using ark-tools.
func NewOptimizedController(config Config) *OptimizedController {
	// Create ark-tools app with initial capacity
	arkApp := app.New(1024)
	world := &arkApp.World
	mapper := entities.NewEntityManager(world)

	// Instantiate the new adaptive queue and dynamic worker pool.
	pulseQueue, err := queue.NewAdaptiveQueue(config.QueueCapacity)
	if err != nil {
		log.Fatalf("Failed to create pulse queue: %v", err)
	}
	interventionQueue, err := queue.NewAdaptiveQueue(config.QueueCapacity)
	if err != nil {
		log.Fatalf("Failed to create intervention queue: %v", err)
	}
	codeQueue, err := queue.NewAdaptiveQueue(config.QueueCapacity)
	if err != nil {
		log.Fatalf("Failed to create code queue: %v", err)
	}

	pulseLogger := log.New(os.Stdout, "[PulsePool] ", log.LstdFlags)
	pulsePool, err := queue.NewDynamicWorkerPool(pulseQueue, config.WorkerConfig, pulseLogger)
	if err != nil {
		log.Fatalf("Failed to create pulse worker pool: %v", err)
	}
	interventionLogger := log.New(os.Stdout, "[InterventionPool] ", log.LstdFlags)
	interventionPool, err := queue.NewDynamicWorkerPool(interventionQueue, config.WorkerConfig,
interventionLogger)
	if err != nil {
		log.Fatalf("Failed to create intervention worker pool: %v", err)
	}
	codeLogger := log.New(os.Stdout, "[CodePool] ", log.LstdFlags)
	codePool, err := queue.NewDynamicWorkerPool(codeQueue, config.WorkerConfig, codeLogger)
	if err != nil {
		log.Fatalf("Failed to create code worker pool: %v", err)
```

```go
	}

	logger := &LoggerAdapter{logger: SystemLogger}

	// Instantiate the refactored systems with dedicated queues and worker pools.
	pulseRouter := pulsePool.GetRouter()
	interventionRouter := interventionPool.GetRouter()
	codeRouter := codePool.GetRouter()

	pulseScheduleSystem := systems.NewBatchPulseScheduleSystem(world, logger)
	pulseSystem := systems.NewBatchPulseSystem(world, pulseQueue, config.BatchSize, logger)
	pulseResultSystem := systems.NewBatchPulseResultSystem(world, pulseRouter.PulseResultChan,
logger)

	interventionSystem := systems.NewBatchInterventionSystem(world, interventionQueue,
config.BatchSize, logger)
	interventionResultSystem := systems.NewBatchInterventionResultSystem(world,
interventionRouter.InterventionResultChan, logger)

	codeSystem := systems.NewBatchCodeSystem(world, codeQueue, config.BatchSize, logger)
	codeResultSystem := systems.NewBatchCodeResultSystem(world, codeRouter.CodeResultChan, logger)

	arkApp.AddSystem(pulseScheduleSystem)
	arkApp.AddSystem(pulseSystem)
	arkApp.AddSystem(interventionSystem)
	arkApp.AddSystem(codeSystem)
	arkApp.AddSystem(pulseResultSystem)
	arkApp.AddSystem(interventionResultSystem)
	arkApp.AddSystem(codeResultSystem)

	return &OptimizedController{
		app:                      arkApp,
		world:                    world,
		mapper:                   mapper,
		pulseQueue:               pulseQueue,
		interventionQueue:        interventionQueue,
		codeQueue:                codeQueue,
		pulsePool:                pulsePool,
		interventionPool:         interventionPool,
		codePool:                 codePool,
		pulseScheduleSystem:      pulseScheduleSystem,
		pulseSystem:              pulseSystem,
		pulseResultSystem:        pulseResultSystem,
		interventionSystem:       interventionSystem,
		interventionResultSystem: interventionResultSystem,
		codeSystem:               codeSystem,
		codeResultSystem:         codeResultSystem,
		config:                   config,
	}
}

// LoadMonitors loads monitors using the streaming loader.
func (c *OptimizedController) LoadMonitors(ctx context.Context, filename string) error {
	loader := streaming.NewStreamingLoader(filename, c.world, c.config.StreamingConfig)
	stats, err := loader.Load(ctx)
	if err != nil {
		return fmt.Errorf("failed to load monitors: %w", err)
	}
	SystemLogger.Info("Successfully loaded %d monitors in %v (%.0f monitors/sec)",
		stats.TotalEntities, stats.LoadingTime, stats.CreationRate)
	if stats.PulseRate > 0 {
		limit := int(math.Ceil(stats.PulseRate * c.config.UpdateInterval.Seconds()))
		if limit < 1 {
			limit = 1
		}
		c.pulseSystem.SetMaxDispatch(limit)
	}
	return nil
}

// Start begins the main processing loop of the controller.
func (c *OptimizedController) Start(ctx context.Context) error {
	if c.running {
		return fmt.Errorf("controller already running")
	}
	c.pulsePool.Start()
	c.interventionPool.Start()
	c.codePool.Start()
	c.running = true
	go c.app.Run()
```

```
        SystemLogger.Info("Optimized controller started successfully")
        return nil
}

// Stop gracefully shuts down the controller.
func (c *OptimizedController) Stop() {
        if !c.running {
                return
        }
        SystemLogger.Info("Stopping controller...")
        c.app.Finalize()
        c.running = false
        c.pulsePool.DrainAndStop()
        c.interventionPool.DrainAndStop()
        c.codePool.DrainAndStop()
        c.PrintShutdownMetrics()
        c.pulseQueue.Close()
        c.interventionQueue.Close()
        c.codeQueue.Close()
        SystemLogger.Info("Controller stopped")
}

// PrintShutdownMetrics logs queue, worker pool, and world statistics at shutdown.
func (c *OptimizedController) PrintShutdownMetrics() {
        logQueue := func(label string, stats queue.Stats) {
                SystemLogger.Info("%s Queue: depth=%d/%d enqueued=%d dequeued=%d dropped=%d", label,
stats.QueueDepth, stats.Capacity, stats.Enqueued, stats.Dequeued, stats.Dropped)
                SystemLogger.Info("%s Queue timings: avg_wait=%v max_wait=%v window=%v", label,
stats.AvgQueueTime, stats.MaxQueueTime, stats.SampleWindow)
                SystemLogger.Info("%s Queue rates: arrival=%.2f/s service=%.2f/s last_enqueue=%v
last_dequeue=%v", label, stats.EnqueueRate, stats.DequeueRate, stats.LastEnqueue, stats.LastDequeue)
        }
        logWorkers := func(label string, stats queue.WorkerPoolStats) {
                SystemLogger.Info("%s Workers: running=%d capacity=%d target=%d min=%d max=%d
waiting=%d", label, stats.RunningWorkers, stats.CurrentCapacity, stats.TargetWorkers,
stats.MinWorkers, stats.MaxWorkers, stats.WaitingTasks)
                SystemLogger.Info("%s Tasks: submitted=%d completed=%d pending_results=%d
scaling_events=%d last_scale=%v", label, stats.TasksSubmitted, stats.TasksCompleted,
stats.PendingResults, stats.ScalingEvents, stats.LastScaleTime)
        }

        SystemLogger.Info("=== SHUTDOWN METRICS ===")

        logQueue("Pulse", c.pulseQueue.Stats())
        logQueue("Intervention", c.interventionQueue.Stats())
        logQueue("Code", c.codeQueue.Stats())

        logWorkers("Pulse", c.pulsePool.Stats())
        logWorkers("Intervention", c.interventionPool.Stats())
        logWorkers("Code", c.codePool.Stats())

        worldStats := c.world.Stats()
        SystemLogger.Info("World: entities_used=%d recycled=%d total=%d archetypes=%d components=%d
filters=%d locked=%t",
                worldStats.Entities.Used, worldStats.Entities.Recycled, worldStats.Entities.Total,
                len(worldStats.Archetypes), len(worldStats.ComponentTypes), worldStats.CachedFilters,
worldStats.Locked)
        SystemLogger.Info("World memory: reserved=%dB used=%dB", worldStats.Memory,
worldStats.MemoryUsed)
        SystemLogger.Info("========================")
}

// GetWorld returns the ECS world for external access (e.g., testing, debugging).
func (c *OptimizedController) GetWorld() *ecs.World {
        return c.world
}


/* controller/recovery.go */

package controller

import (
        "cpra/internal/controller/entities"
        "log"
        "runtime/debug"
        "time"

        "github.com/mlange-42/ark/ecs"
)
```

```
// RecoverySystem provides system-level error recovery and health monitoring
type RecoverySystem struct {
	ErrorCount  int
	LastError   time.Time
	MaxErrors   int
	ResetWindow time.Duration
	Mapper      *entities.EntityManager
}

func NewRecoverySystem(maxErrors int, resetWindow time.Duration) *RecoverySystem {
	return &RecoverySystem{
		MaxErrors:   maxErrors,
		ResetWindow: resetWindow,
	}
}

// SafeSystemUpdate wraps system updates with error recovery
func (r *RecoverySystem) SafeSystemUpdate(systemName string, updateFunc func() error) error {
	defer func() {
		if recovered := recover(); recovered != nil {
			r.ErrorCount++
			r.LastError = time.Now()

			log.Printf("PANIC in system %s: %v", systemName, recovered)
			log.Printf("Stack trace: %s", debug.Stack())

			// Circuit breaker logic
			if r.ErrorCount >= r.MaxErrors {
				log.Printf("System %s exceeded max errors (%d), entering degraded
mode",
					systemName, r.MaxErrors)
			}
		}
	}()

	// Reset error count if enough time has passed
	if time.Since(r.LastError) > r.ResetWindow {
		r.ErrorCount = 0
	}

	// Circuit breaker - prevent further damage if too many errors
	if r.ErrorCount >= r.MaxErrors {
		return nil // Skip execution
	}

	return updateFunc()
}

// ValidateEntityHealth checks entity component integrity
func (r *RecoverySystem) ValidateEntityHealth(w *ecs.World, entity ecs.Entity) bool {
	if !w.Alive(entity) {
		return false
	}

	// Check for required components
	if r.Mapper != nil {
		state := r.Mapper.GetMonitorState(entity)
		if state == nil {
			log.Printf("Entity %v missing MonitorState component", entity)
			return false
		}
		if state.Name == "" {
			log.Printf("Entity %v missing Name component", entity)
			return false
		}
	}

	return true
}

// CleanupOrphanedComponents removes components from dead entities
func (r *RecoverySystem) CleanupOrphanedComponents(w *ecs.World) {
	// This would need specific implementation based on component tracking
	// For now, log the cleanup intent
	log.Printf("Cleanup cycle: %d entities active", w.Stats().Entities.Used)
}


/* controller/systems/batch_code_result_system.go */
```

```go
package systems

import (
	"cpra/internal/controller/components"
	"cpra/internal/jobs"
	"sync/atomic"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// BatchCodeResultSystem processes the results of dispatched code alerts.
// It processes batches of results passed directly from the result router.
type BatchCodeResultSystem struct {
	world  *ecs.World
	logger Logger

	// Mappers for efficient component access
	stateMapper *ecs.Map[components.MonitorState]
	ResultChan  <-chan []jobs.Result
}

// NewBatchCodeResultSystem creates a new BatchCodeResultSystem.
func NewBatchCodeResultSystem(world *ecs.World, results <-chan []jobs.Result, logger Logger)
*BatchCodeResultSystem {
	return &BatchCodeResultSystem{
		world:       world,
		logger:      logger,
		stateMapper: ecs.NewMap[components.MonitorState](world),
		ResultChan:  results,
	}
}

func (s *BatchCodeResultSystem) Initialize(w *ecs.World) {
}

func (s *BatchCodeResultSystem) Update(w *ecs.World) {
	if s.ResultChan == nil {
		return
	}

	resultsBatches := make([][]jobs.Result, 0)
loop:
	for {
		select {
		case res, ok := <-s.ResultChan:
			if !ok {
				s.ResultChan = nil
				break loop
			}
			if len(res) == 0 {
				continue
			}
			resultsBatches = append(resultsBatches, res)
		default:
			break loop
		}
	}

	for _, res := range resultsBatches {
		s.ProcessBatch(res)
	}
}

// ProcessBatch processes a batch of code alert results.
func (s *BatchCodeResultSystem) ProcessBatch(results []jobs.Result) {
	startTime := time.Now()
	processedCount := 0

	for _, result := range results {
		ent := result.Entity()
		if !s.world.Alive(ent) {
			continue
		}

		state := s.stateMapper.Get(ent)
		if state == nil {
			continue
		}
```

```go
		// Ensure we are processing a pending code alert.
		if (atomic.LoadUint32(&state.Flags) & components.StateCodePending) == 0 {
			s.logger.Warn("Entity[%d] received CodeResult but was not in CodePending
state", ent.ID())
			continue
		}

		processedCount++

		// Extract color from the result payload.
		colorPayload, ok := result.Payload["color"]
		if !ok {
			s.logger.Warn("Entity[%d] has CodeResult with no color in payload", ent.ID())
			continue
		}
		color, ok := colorPayload.(string)
		if !ok {
			s.logger.Warn("Entity[%d] has CodeResult with invalid color payload type",
ent.ID())
			continue
		}

		if err := result.Error(); err != nil {
			s.logger.Error("Monitor '%s' %s alert failed to send: %v", state.Name, color,
err)
		} else {
			s.logger.Info("Monitor '%s' %s alert sent successfully.", state.Name, color)
		}

		// Unset the pending flag.
		atomic.AndUint32(&state.Flags, ^uint32(components.StateCodePending))
	}

	if processedCount > 0 {
		s.logger.LogSystemPerformance("BatchCodeResultSystem", time.Since(startTime),
processedCount)
	}
}

// Finalize is a no-op for this system.
func (s *BatchCodeResultSystem) Finalize(w *ecs.World) {}


/* controller/systems/batch_code_system.go */

package systems

import (
	"cpra/internal/controller/components"
	"cpra/internal/jobs"
	"cpra/internal/queue"
	"reflect"
	"sync/atomic"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// jobInfo is a helper struct to associate a job with its entity and color for batch processing.
type jobInfo struct {
	Entity ecs.Entity
	Job    jobs.Job
	Color  string
}

// BatchCodeSystem processes entities that need a code alert dispatched.
// It determines the correct color based on the entity's state and enqueues the job.
type BatchCodeSystem struct {
	world     *ecs.World
	queue     queue.Queue // Using a generic queue interface
	logger    Logger
	batchSize int

	// Filter for entities that require a code alert.
	filter      *ecs.Filter3[components.MonitorState, components.CodeConfig,
components.JobStorage]
	stateMapper *ecs.Map1[components.MonitorState]
}
```

```
// NewBatchCodeSystem creates a new BatchCodeSystem.
func NewBatchCodeSystem(world *ecs.World, q queue.Queue, batchSize int, logger Logger)
*BatchCodeSystem {
	return &BatchCodeSystem{
		world:       world,
		queue:       q,
		logger:      logger,
		batchSize:   batchSize,
		filter:      ecs.NewFilter3[components.MonitorState, components.CodeConfig,
components.JobStorage](world),
		stateMapper: ecs.NewMap1[components.MonitorState](world),
	}
}
func (s *BatchCodeSystem) Initialize(w *ecs.World) {

}

// Update finds and processes all monitors that need a code alert.
func (s *BatchCodeSystem) Update(w *ecs.World) {
	startTime := time.Now()
	stats := s.queue.Stats()
	if stats.Capacity > 0 && stats.QueueDepth >= int(float64(stats.Capacity)*0.9) {
		s.logger.Debug("Code queue saturated (%d/%d); deferring dispatch", stats.QueueDepth,
stats.Capacity)
	}

	query := s.filter.Query()

	free := stats.Capacity - stats.QueueDepth
	if free <= 0 {
		return
	}
	tokens := int(float64(free) * 0.8)
	if tokens <= 0 {
		tokens = free
	}
	if tokens <= 0 {
		tokens = 1
	}

	earlyExit := false

	jobsToProcess := make([]jobInfo, 0, s.batchSize)
	processedCount := 0

	for query.Next() {
		ent := query.Entity()
		state, _, jobStorage := query.Get()

		// Process only entities that need a code alert.
		if (atomic.LoadUint32(&state.Flags) & components.StateCodeNeeded) == 0 {
			continue
		}

		color := state.PendingCode
		if color == "" {
			// This should not happen if StateCodeNeeded is set, but as a safeguard:
			atomic.AndUint32(&state.Flags, ^uint32(components.StateCodeNeeded))
			continue
		}

		job, ok := jobStorage.CodeJobs[color]
		if !ok || isNilJob(job) {
			s.logger.Warn("Entity[%d] needs '%s' code alert, but no job is configured.",
ent.ID(), color)
			// Clear the flag if no job is found to prevent spinning.
			atomic.AndUint32(&state.Flags, ^uint32(components.StateCodeNeeded))
			continue
		}

		jobsToProcess = append(jobsToProcess, jobInfo{Entity: ent, Job: job, Color: color})

		if len(jobsToProcess) >= tokens {
			s.processBatch(&jobsToProcess)
			processedCount += len(jobsToProcess)
			jobsToProcess = make([]jobInfo, 0, s.batchSize)
			earlyExit = true
			break
		}
	}
```

```go
	// Process any remaining entities
	if earlyExit {
		query.Close()
	}

	if len(jobsToProcess) > 0 {
		s.processBatch(&jobsToProcess)
		processedCount += len(jobsToProcess)
	}

	if processedCount > 0 {
		s.logger.LogSystemPerformance("BatchCodeSystem", time.Since(startTime),
processedCount)
	}

}

// processBatch attempts to enqueue a batch of jobs and updates entity states on success.
func (s *BatchCodeSystem) processBatch(jobsInfo *[]jobInfo) {
	stats := s.queue.Stats()
	if stats.Capacity > 0 && stats.QueueDepth >= int(float64(stats.Capacity)*0.9) {
		s.logger.Debug("Code queue near capacity (%d/%d); skipping enqueue", stats.QueueDepth,
stats.Capacity)
		return
	}
	jobs := make([]interface{}, 0, len(*jobsInfo))
	submitted := make([]jobInfo, 0, len(*jobsInfo))
	for _, info := range *jobsInfo {
		if isNilJob(info.Job) {
			s.logger.Warn("Entity[%d] code job became nil before enqueue; skipping",
info.Entity.ID())
			continue
		}
		jobs = append(jobs, info.Job)
		submitted = append(submitted, info)
	}

	if len(jobs) == 0 {
		return
	}

	err := s.queue.EnqueueBatch(jobs)
	if err != nil {
		s.logger.Warn("Failed to enqueue code job batch, queue may be full: %v", err)
		return
	}

	for _, info := range submitted {
		if !s.world.Alive(info.Entity) {
			continue
		}
		state := s.stateMapper.Get(info.Entity)
		if state == nil {
			continue
		}

		for {
			flags := atomic.LoadUint32(&state.Flags)
			if flags&components.StateCodeNeeded == 0 {
				break
			}

			updated := (flags & ^uint32(components.StateCodeNeeded)) |
uint32(components.StateCodePending)
			if atomic.CompareAndSwapUint32(&state.Flags, flags, updated) {
				state.PendingCode = ""
				s.logger.Info("CODE DISPATCHED: %s (%s)", state.Name, info.Color)
				break
			}
		}
	}
}

// Finalize is a no-op for this system.
func (s *BatchCodeSystem) Finalize(w *ecs.World) {}

func isNilJob(job jobs.Job) bool {
	if job == nil {
		return true
```

```go
	}
	v := reflect.ValueOf(job)
	switch v.Kind() {
	case reflect.Ptr, reflect.Interface, reflect.Slice, reflect.Map, reflect.Func, reflect.Chan:
		return v.IsNil()
	default:
		return false
	}
}


/* controller/systems/batch_intervention_result_system.go */

package systems

import (
	"cpra/internal/controller/components"
	"cpra/internal/jobs"
	"sync/atomic"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// BatchInterventionResultSystem processes completed intervention jobs.
// It processes batches of results passed directly from the result router.
type BatchInterventionResultSystem struct {
	world  *ecs.World
	logger Logger

	// Mappers for efficient component access
	stateMapper *ecs.Map[components.MonitorState]
	ResultChan  <-chan []jobs.Result
}

// NewBatchInterventionResultSystem creates a new BatchInterventionResultSystem.
func NewBatchInterventionResultSystem(world *ecs.World, results <-chan []jobs.Result, logger Logger)
*BatchInterventionResultSystem {
	return &BatchInterventionResultSystem{
		world:       world,
		logger:      logger,
		stateMapper: ecs.NewMap[components.MonitorState](world),
		ResultChan:  results,
	}
}

func (s *BatchInterventionResultSystem) Initialize(w *ecs.World) {
}

func (s *BatchInterventionResultSystem) Update(w *ecs.World) {
	if s.ResultChan == nil {
		return
	}

	resultsBatches := make([][]jobs.Result, 0)
loop:
	for {
		select {
		case res, ok := <-s.ResultChan:
			if !ok {
				s.ResultChan = nil
				break loop
			}
			if len(res) == 0 {
				continue
			}
			resultsBatches = append(resultsBatches, res)
		default:
			break loop
		}
	}

	for _, res := range resultsBatches {
		s.ProcessBatch(res)
	}
}

// ProcessBatch processes a batch of intervention results.
func (s *BatchInterventionResultSystem) ProcessBatch(results []jobs.Result) {
	startTime := time.Now()
```

```
	processedCount := 0

	for _, result := range results {
		ent := result.Entity()
		if !s.world.Alive(ent) {
			continue
		}

		state := s.stateMapper.Get(ent)
		if state == nil {
			continue
		}

		// Ensure we are processing a pending intervention
		if (atomic.LoadUint32(&state.Flags) & components.StateInterventionPending) == 0 {
			s.logger.Warn("Entity[%d] received InterventionResult but was not in
InterventionPending state", ent.ID())
			continue
		}

		processedCount++
		state.LastCheckTime = time.Now()

		if result.Error() != nil {
			// --- FAILURE ---
			state.ConsecutiveFailures++
			state.LastError = result.Error()
			s.logger.Error("Monitor '%s' intervention failed: %v", state.Name,
state.LastError)
			s.triggerCode(ent, state, "red")
		} else {
			// --- SUCCESS ---
			s.logger.Info("Monitor '%s' intervention succeeded.", state.Name)
			state.ConsecutiveFailures = 0
			state.LastError = nil
			state.LastSuccessTime = state.LastCheckTime
			s.triggerCode(ent, state, "cyan")
		}

		// Unset the pending flag, regardless of outcome.
		atomic.AndUint32(&state.Flags, ^uint32(components.StateInterventionPending))
	}

	if processedCount > 0 {
		s.logger.LogSystemPerformance("BatchInterventionResultSystem", time.Since(startTime),
processedCount)
	}
}

func (s *BatchInterventionResultSystem) triggerCode(entity ecs.Entity, state *components.MonitorState,
color string) {
	codeConfigMapper := ecs.NewMap[components.CodeConfig](s.world)
	if !codeConfigMapper.Has(entity) {
		return
	}
	codeConfig := codeConfigMapper.Get(entity)
	if _, ok := codeConfig.Configs[color]; ok {
		state.PendingCode = color
		atomic.OrUint32(&state.Flags, components.StateCodeNeeded)
		s.logger.Info("Monitor '%s' - flagging for %s alert code", state.Name, color)
	}
}

// Finalize is a no-op for this system.
func (s *BatchInterventionResultSystem) Finalize(w *ecs.World) {}


/* controller/systems/batch_intervention_system.go */

package systems

import (
	"cpra/internal/controller/components"
	"cpra/internal/queue"
	"sync/atomic"
	"time"

	"github.com/mlange-42/ark/ecs"
)
```

```
// BatchInterventionSystem processes entities that need an intervention.
// It identifies entities with the StateInterventionNeeded flag, enqueues the corresponding job,
// and transitions the entity state to StateInterventionPending.
type BatchInterventionSystem struct {
	world     *ecs.World
	queue     queue.Queue // Using a generic queue interface
	logger    Logger
	batchSize int

	// Filter for entities that require an intervention.
	filter             *ecs.Filter3[components.MonitorState, components.InterventionConfig,
components.JobStorage]
	monitorStateMapper *ecs.Map[components.MonitorState]
}

// NewBatchInterventionSystem creates a new BatchInterventionSystem.
func NewBatchInterventionSystem(world *ecs.World, q queue.Queue, batchSize int, logger Logger)
*BatchInterventionSystem {
	return &BatchInterventionSystem{
		world:              world,
		queue:              q,
		logger:             logger,
		batchSize:          batchSize,
		filter:             ecs.NewFilter3[components.MonitorState,
components.InterventionConfig, components.JobStorage](world),
		monitorStateMapper: ecs.NewMap[components.MonitorState](world),
	}
}

func (s *BatchInterventionSystem) Initialize(w *ecs.World) {}

// Update finds and processes all monitors that need an intervention.
func (s *BatchInterventionSystem) Update(w *ecs.World) {
	startTime := time.Now()
	stats := s.queue.Stats()
	if stats.Capacity > 0 && stats.QueueDepth >= int(float64(stats.Capacity)*0.9) {
		s.logger.Debug("Intervention queue saturated (%d/%d); deferring dispatch",
stats.QueueDepth, stats.Capacity)
	}

	query := s.filter.Query()

	free := stats.Capacity - stats.QueueDepth
	if free <= 0 {
		return
	}
	tokens := int(float64(free) * 0.8)
	if tokens <= 0 {
		tokens = free
	}
	if tokens <= 0 {
		tokens = 1
	}

	earlyExit := false

	jobsToQueue := make([]interface{}, 0, s.batchSize)
	entitiesToUpdate := make([]ecs.Entity, 0, s.batchSize)
	processedCount := 0

	for query.Next() {
		ent := query.Entity()
		state, _, jobStorage := query.Get()

		// Process only entities that need an intervention.
		if (atomic.LoadUint32(&state.Flags) & components.StateInterventionNeeded) == 0 {
			continue
		}

		if jobStorage.InterventionJob == nil {
			s.logger.Warn("Entity[%d] has InterventionNeeded state but no
InterventionJob", ent.ID())
			continue
		}

		jobsToQueue = append(jobsToQueue, jobStorage.InterventionJob)
		entitiesToUpdate = append(entitiesToUpdate, ent)

		if len(jobsToQueue) >= tokens {
			s.processBatch(&jobsToQueue, &entitiesToUpdate)
```

```go
                        processedCount += len(jobsToQueue)
                        jobsToQueue = make([]interface{}, 0, s.batchSize)
                        entitiesToUpdate = make([]ecs.Entity, 0, s.batchSize)
                        earlyExit = true
                        break
                }
        }

        // Process any remaining entities
        if earlyExit {
                query.Close()
        }

        if len(jobsToQueue) > 0 {
                s.processBatch(&jobsToQueue, &entitiesToUpdate)
                processedCount += len(jobsToQueue)
        }

        if processedCount > 0 {
                s.logger.LogSystemPerformance("BatchInterventionSystem", time.Since(startTime), 
processedCount)
        }

}

// processBatch attempts to enqueue a batch of jobs and updates entity states on success.
func (s *BatchInterventionSystem) processBatch(jobs *[]interface{}, entities *[]ecs.Entity) {
        stats := s.queue.Stats()
        if stats.Capacity > 0 && stats.QueueDepth >= int(float64(stats.Capacity)*0.9) {
                s.logger.Debug("Intervention queue near capacity (%d/%d); skipping enqueue", 
stats.QueueDepth, stats.Capacity)
                return
        }
        err := s.queue.EnqueueBatch(*jobs)
        if err != nil {
                s.logger.Warn("Failed to enqueue intervention job batch, queue may be full: %v", err)
                // Do not transition state if enqueue fails, allowing retry on the next tick.
                return
        }

        // If enqueue is successful, transition the state for all entities in the batch.
        for _, ent := range *entities {
                if !s.world.Alive(ent) {
                        continue
                }
                state := s.monitorStateMapper.Get(ent)
                if state == nil {
                        continue
                }

                for {
                        flags := atomic.LoadUint32(&state.Flags)
                        if flags&components.StateInterventionNeeded == 0 {
                                break
                        }

                        updated := (flags & ^uint32(components.StateInterventionNeeded)) | 
uint32(components.StateInterventionPending)
                        if atomic.CompareAndSwapUint32(&state.Flags, flags, updated) {
                                s.logger.Info("INTERVENTION DISPATCHED: %s", state.Name)
                                break
                        }
                }
        }
}

// Finalize is a no-op for this system.
func (s *BatchInterventionSystem) Finalize(w *ecs.World) {}


/* controller/systems/batch_pulse_result_system.go */

package systems

import (
        "cpra/internal/controller/components"
        "cpra/internal/jobs"
        "sync/atomic"
        "time"
```

```go
	"github.com/mlange-42/ark/ecs"
)

// BatchPulseResultSystem processes completed pulse checks.
// It queries for entities with a PulseResult component and updates their state accordingly.
type BatchPulseResultSystem struct {
	world  *ecs.World
	logger Logger

	// Mappers are used for efficient component access
	stateMapper      *ecs.Map1[components.MonitorState]
	configMapper     *ecs.Map1[components.PulseConfig]
	codeConfigMapper *ecs.Map1[components.CodeConfig]
	ResultChan       <-chan []jobs.Result
}

// NewBatchPulseResultSystem creates a new BatchPulseResultSystem.
func NewBatchPulseResultSystem(world *ecs.World, results <-chan []jobs.Result, logger Logger)
*BatchPulseResultSystem {
	return &BatchPulseResultSystem{
		world:            world,
		logger:           logger,
		stateMapper:      ecs.NewMap1[components.MonitorState](world),
		configMapper:     ecs.NewMap1[components.PulseConfig](world),
		codeConfigMapper: ecs.NewMap1[components.CodeConfig](world),
		ResultChan:       results,
	}
}
func (s *BatchPulseResultSystem) Initialize(w *ecs.World) {
}

func (s *BatchPulseResultSystem) Update(w *ecs.World) {
	if s.ResultChan == nil {
		return
	}

	resultsBatches := make([][]jobs.Result, 0)
loop:
	for {
		select {
		case res, ok := <-s.ResultChan:
			if !ok {
				s.ResultChan = nil
				break loop
			}
			resultsBatches = append(resultsBatches, res)
		default:
			break loop
		}
	}

	for _, res := range resultsBatches {
		s.ProcessBatch(res)
	}
}

// ProcessBatch processes a batch of pulse results.
func (s *BatchPulseResultSystem) ProcessBatch(results []jobs.Result) {
	startTime := time.Now()
	processedCount := 0

	for _, result := range results {
		ent := result.Entity()
		if !s.world.Alive(ent) {
			continue
		}

		state := s.stateMapper.Get(ent)
		config := s.configMapper.Get(ent)

		if (atomic.LoadUint32(&state.Flags) & components.StatePulsePending) == 0 {
			s.logger.Warn("Entity[%d] received a PulseResult but was not in a PulsePending
state.", ent.ID())
			continue
		}

		processedCount++
		state.LastCheckTime = time.Now()

		if result.Error() != nil {
```

```
			// --- FAILURE ---
			state.ConsecutiveFailures++
			state.LastError = result.Error()
			s.logger.Warn("Monitor '%s' pulse failed (%d/%d): %v", state.Name,
state.ConsecutiveFailures, config.MaxFailures, state.LastError)

			if state.ConsecutiveFailures >= config.MaxFailures {
				s.logger.Warn("Monitor '%s' reached max failures, triggering
intervention.", state.Name)
				atomic.OrUint32(&state.Flags, components.StateInterventionNeeded)
				state.ConsecutiveFailures = 0 // Reset after triggering
			} else if state.ConsecutiveFailures == 1 {
				s.triggerCode(ent, state, "yellow")
			}
		} else {
			// --- SUCCESS ---
			wasFailure := state.ConsecutiveFailures > 0
			if wasFailure {
				s.logger.Info("Monitor '%s' pulse recovered.", state.Name)
				s.triggerCode(ent, state, "green")
			}
			state.ConsecutiveFailures = 0
			state.LastError = nil
			state.LastSuccessTime = state.LastCheckTime
		}

		// Unset the pending flag, regardless of outcome.
		atomic.AndUint32(&state.Flags, ^uint32(components.StatePulsePending))
	}

	if processedCount > 0 {
		s.logger.LogSystemPerformance("BatchPulseResultSystem", time.Since(startTime),
processedCount)
	}
}

func (s *BatchPulseResultSystem) triggerCode(entity ecs.Entity, state *components.MonitorState, color
string) {
	codeConfig := s.codeConfigMapper.Get(entity)
	if codeConfig == nil {
		return
	}
	if _, ok := codeConfig.Configs[color]; ok {
		// TODO: This is a placeholder for a more robust CodeNeeded implementation
		// For now, we directly set the flag.
		state.PendingCode = color
		atomic.OrUint32(&state.Flags, components.StateCodeNeeded)
		s.logger.Info("Monitor '%s' - triggering %s alert code", state.Name, color)
	}
}

// Finalize is a no-op for this system.
func (s *BatchPulseResultSystem) Finalize(w *ecs.World) {}


/* controller/systems/batch_pulse_schedule_system.go */

package systems

import (
	"sync/atomic"
	"time"

	"cpra/internal/controller/components"
	"github.com/mlange-42/ark/ecs"
)

// BatchPulseScheduleSystem schedules pulse checks for entities that are due.
// It queries for monitors that are not disabled, not already pending a pulse check,
// and whose interval has passed since the last check.
// This system is a critical part of the monitoring pipeline, ensuring that checks
// are scheduled in a timely and efficient manner.
type BatchPulseScheduleSystem struct {
	world  *ecs.World
	logger Logger

	// Filter for entities that are candidates for a pulse check.
	filter *ecs.Filter2[components.MonitorState, components.PulseConfig]
}
```

```go
// NewBatchPulseScheduleSystem creates a new BatchPulseScheduleSystem.
func NewBatchPulseScheduleSystem(world *ecs.World, logger Logger) *BatchPulseScheduleSystem {
	return &BatchPulseScheduleSystem{
		world:  world,
		logger: logger,
		filter: ecs.NewFilter2[components.MonitorState, components.PulseConfig](world),
	}
}

func (s *BatchPulseScheduleSystem) Initialize(w *ecs.World) {

}

// Update finds and schedules all monitors that are due for a pulse check.
func (s *BatchPulseScheduleSystem) Update(w *ecs.World) {
	start := time.Now()
	query := s.filter.Query()
	var scheduledCount int

	now := time.Now()

	for query.Next() {
		state, config := query.Get()

		for {
			flags := atomic.LoadUint32(&state.Flags)

			if (flags&components.StateDisabled != 0) || (flags&components.StatePulseNeeded
!= 0) || (flags&components.StatePulsePending != 0) {
				break
			}

			due := (flags&components.StatePulseFirstCheck != 0) ||
(now.Sub(state.LastCheckTime) >= config.Interval)
			if !due {
				break
			}

			updated := (flags | components.StatePulseNeeded) &^
components.StatePulseFirstCheck
			if atomic.CompareAndSwapUint32(&state.Flags, flags, updated) {
				scheduledCount++
				break
			}
		}
	}

	if scheduledCount > 0 {
		s.logger.LogSystemPerformance("BatchPulseScheduleSystem", time.Since(start),
scheduledCount)
	}

}

// Finalize is a no-op for this system.
func (s *BatchPulseScheduleSystem) Finalize(w *ecs.World) {
	// Nothing to clean up
}


/* controller/systems/batch_pulse_system.go */

package systems

import (
	"cpra/internal/controller/components"
	"cpra/internal/queue"
	"sync/atomic"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// BatchPulseSystem processes entities that need a pulse check.
// It identifies entities with the StatePulseNeeded flag, enqueues the corresponding job,
// and transitions the entity state to StatePulsePending.
type BatchPulseSystem struct {
	world       *ecs.World
	queue       queue.Queue // Using a generic queue interface
	logger      Logger
```

```go
	batchSize   int
	maxDispatch int

	// Filter for entities that require a pulse check.
	filter             *ecs.Filter2[components.MonitorState, components.JobStorage]
	monitorStateMapper *ecs.Map[components.MonitorState]
}

// NewBatchPulseSystem creates a new BatchPulseSystem.
func NewBatchPulseSystem(world *ecs.World, q queue.Queue, batchSize int, logger Logger)
*BatchPulseSystem {
	return &BatchPulseSystem{
		world:              world,
		queue:              q,
		logger:             logger,
		batchSize:          batchSize,
		filter:             ecs.NewFilter2[components.MonitorState, components.JobStorage]
(world),
		monitorStateMapper: ecs.NewMap[components.MonitorState](world),
	}
}

func (s *BatchPulseSystem) Initialize(w *ecs.World) {

}

func (s *BatchPulseSystem) SetMaxDispatch(n int) {
	s.maxDispatch = n
}

// Update finds and processes all monitors that need a pulse check.
func (s *BatchPulseSystem) Update(w *ecs.World) {
	startTime := time.Now()
	stats := s.queue.Stats()
	if stats.Capacity > 0 && stats.QueueDepth >= int(float64(stats.Capacity)*0.9) {
		s.logger.Debug("Pulse queue saturated (%d/%d); deferring dispatch", stats.QueueDepth,
stats.Capacity)
	}

	query := s.filter.Query()
	free := stats.Capacity - stats.QueueDepth
	if free <= 0 {
		return
	}
	tokens := int(float64(free) * 0.8)
	if tokens <= 0 {
		tokens = free
	}
	if tokens <= 0 {
		tokens = 1
	}
	if s.maxDispatch > 0 && tokens > s.maxDispatch {
		tokens = s.maxDispatch
	}

	earlyExit := false

	jobsToQueue := make([]interface{}, 0, s.batchSize)
	entitiesToUpdate := make([]ecs.Entity, 0, s.batchSize)
	processedCount := 0

	for query.Next() {
		ent := query.Entity()
		state, jobStorage := query.Get()

		// Process only entities that need a pulse check.
		if (atomic.LoadUint32(&state.Flags) & components.StatePulseNeeded) == 0 {
			continue
		}

		if jobStorage.PulseJob == nil {
			s.logger.Warn("Entity[%d] has PulseNeeded state but no PulseJob", ent.ID())
			continue
		}

		jobsToQueue = append(jobsToQueue, jobStorage.PulseJob)
		entitiesToUpdate = append(entitiesToUpdate, ent)

		if len(jobsToQueue) >= tokens {
			s.processBatch(&jobsToQueue, &entitiesToUpdate)
```

```
				processedCount += len(jobsToQueue)
				jobsToQueue = make([]interface{}, 0, s.batchSize)
				entitiesToUpdate = make([]ecs.Entity, 0, s.batchSize)
				earlyExit = true
				break
			}
		}

		// Process any remaining entities
		if earlyExit {
			query.Close()
		}

		if len(jobsToQueue) > 0 {
			s.processBatch(&jobsToQueue, &entitiesToUpdate)
			processedCount += len(jobsToQueue)
		}

		if processedCount > 0 {
			s.logger.LogSystemPerformance("BatchPulseSystem", time.Since(startTime),
processedCount)
		}

}

// processBatch attempts to enqueue a batch of jobs and updates entity states on success.
func (s *BatchPulseSystem) processBatch(jobs *[]interface{}, entities *[]ecs.Entity) {
		stats := s.queue.Stats()
		if stats.Capacity > 0 && stats.QueueDepth >= int(float64(stats.Capacity)*0.9) {
			s.logger.Debug("Pulse queue near capacity (%d/%d); skipping enqueue",
stats.QueueDepth, stats.Capacity)
			return
		}
		err := s.queue.EnqueueBatch(*jobs)
		if err != nil {
			s.logger.Warn("Failed to enqueue pulse job batch, queue may be full: %v", err)
			// Do not transition state if enqueue fails, allowing retry on the next tick.
			return
		}

		// If enqueue is successful, transition the state for all entities in the batch.
		now := time.Now()
		for _, ent := range *entities {
			if !s.world.Alive(ent) {
				continue
			}
			state := s.monitorStateMapper.Get(ent)
			if state == nil {
				continue
			}

			for {
				flags := atomic.LoadUint32(&state.Flags)
				if flags&components.StatePulseNeeded == 0 {
					break
				}

				updated := (flags & ^uint32(components.StatePulseNeeded)) |
uint32(components.StatePulsePending)
				if atomic.CompareAndSwapUint32(&state.Flags, flags, updated) {
					state.LastCheckTime = now
					break
				}
			}
		}
}

// Finalize is a no-op for this system.
func (s *BatchPulseSystem) Finalize(w *ecs.World) {}


/* controller/systems/errors.go */

package systems

import (
		"fmt"
		"time"
)
```

```go
type ErrNoPulseJob struct {
	pulseType string
}

type ErrPulseJobTimeout struct {
	PulseType string
	Timeout   time.Duration
	Retries   int
	Err       error
}

func (e *ErrPulseJobTimeout) Error() string {
	return fmt.Sprintf("Job timeout for pulse type %s after %s (retried %d times): %s",
e.PulseType, e.Timeout, e.Retries, e.Err)
}


/* controller/systems/logger.go */

package systems

import "time"

// Logger interface for structured logging that all optimized systems can use
type Logger interface {
	Info(format string, args ...interface{})
	Debug(format string, args ...interface{})
	Warn(format string, args ...interface{})
	Error(format string, args ...interface{})
	LogSystemPerformance(name string, duration time.Duration, count int)
	LogComponentState(entityID uint32, component string, action string)
}


/* controller/systems/memory_efficient_system.go */

package systems

import (
	"fmt"
	"runtime"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// MemoryEfficientSystem provides memory optimization utilities for ECS systems
type MemoryEfficientSystem struct {
	world           *ecs.World
	gcInterval      time.Duration
	lastGC          time.Time
	memoryThreshold int64

	// Memory statistics
	allocsBefore uint64
	allocsAfter  uint64
	gcCount      uint32
}

// MemoryConfig holds memory management configuration
type MemoryConfig struct {
	GCInterval      time.Duration // How often to check memory
	MemoryThreshold int64         // Memory threshold for forced GC (bytes)
	EnableProfiling bool          // Enable memory profiling
}

// MemoryStats holds memory statistics
type MemoryStats struct {
	Alloc        uint64        // Current allocated memory
	TotalAlloc   uint64        // Total allocated memory
	Sys          uint64        // System memory
	GCCount      uint32        // Number of GC runs
	LastGCTime   time.Time     // Last GC time
	GCPauseTotal time.Duration // Total GC pause time
}

// NewMemoryEfficientSystem creates a new memory management system
func NewMemoryEfficientSystem(world *ecs.World, config MemoryConfig) *MemoryEfficientSystem {
	return &MemoryEfficientSystem{
		world:           world,
```

```go
		gcInterval:      config.GCInterval,
		memoryThreshold: config.MemoryThreshold,
		lastGC:          time.Now(),
	}
}

// Update performs memory management tasks
func (mes *MemoryEfficientSystem) Update() {
	now := time.Now()

	// Check if it's time for memory management
	if now.Sub(mes.lastGC) < mes.gcInterval {
		return
	}

	// Get current memory stats
	var m runtime.MemStats
	runtime.ReadMemStats(&m)

	// Force GC if memory usage is high
	if int64(m.Alloc) > mes.memoryThreshold {
		mes.allocsBefore = m.Alloc
		runtime.GC()

		// Get stats after GC
		runtime.ReadMemStats(&m)
		mes.allocsAfter = m.Alloc
		mes.gcCount++

		fmt.Printf("Forced GC: Memory %d MB -> %d MB (freed %d MB)\n",
			mes.allocsBefore/1024/1024,
			m.Alloc/1024/1024,
			(mes.allocsBefore-m.Alloc)/1024/1024)
	}

	mes.lastGC = now
}

// OptimizeEntityStorage optimizes entity storage by removing unused entities
func (mes *MemoryEfficientSystem) OptimizeEntityStorage() {
	// This would implement entity defragmentation if Ark supports it
	// For now, we just track the optimization request
	fmt.Println("Entity storage optimization requested (not implemented in Ark)")
}

// GetMemoryStats returns current memory statistics
func (mes *MemoryEfficientSystem) GetMemoryStats() MemoryStats {
	var m runtime.MemStats
	runtime.ReadMemStats(&m)

	return MemoryStats{
		Alloc:        m.Alloc,
		TotalAlloc:   m.TotalAlloc,
		Sys:          m.Sys,
		GCCount:      mes.gcCount,
		LastGCTime:   mes.lastGC,
		GCPauseTotal: time.Duration(m.PauseTotalNs),
	}
}

// ForceGC forces garbage collection immediately
func (mes *MemoryEfficientSystem) ForceGC() {
	var m runtime.MemStats
	runtime.ReadMemStats(&m)
	mes.allocsBefore = m.Alloc

	runtime.GC()

	runtime.ReadMemStats(&m)
	mes.allocsAfter = m.Alloc
	mes.gcCount++
	mes.lastGC = time.Now()

	fmt.Printf("Manual GC: Memory %d MB -> %d MB (freed %d MB)\n",
		mes.allocsBefore/1024/1024,
		m.Alloc/1024/1024,
		(mes.allocsBefore-m.Alloc)/1024/1024)
}

// SetMemoryThreshold updates the memory threshold for automatic GC
```

```go
func (mes *MemoryEfficientSystem) SetMemoryThreshold(threshold int64) {
	mes.memoryThreshold = threshold
}

// GetGCCount returns the number of forced GC runs
func (mes *MemoryEfficientSystem) GetGCCount() uint32 {
	return mes.gcCount
}


/* controller/systems/sys_test.go */

package systems

//
//import (
//	"cpra/internal/controller"
//	"cpra/internal/controller/components"
//	"cpra/internal/loader/loader"
//	"log"
//	"testing"
//)
//
//func TestPulseSystem_SchedulesJobs(t *testing.T) {
//	// Arrange
//	l := loader.NewLoader("yaml", "internal/loader/test.yaml.bak")
//	l.Load()
//	m := l.GetManifest()
//	w, err := controller.NewCPRaWorld(&m)
//	if err != nil {
//		log.Fatal(err)
//	}
//	entity := w.Mapper.CreateEntityFromMonitor(m)
//	w.AddComponent(entity, components.PulseConfig{ /* ... */ })
//	w.AddComponent(entity, components.Status{ /* ... */ })
//	// ... setup as needed
//
//	// Act
//	systems.PulseSystem{}.Update(w) // or .Step(w), etc.
//
//	// Assert
//	// Check if job scheduled, status updated, etc.
//	// Use t.Errorf or testify/assert
//}


/* controller/tracing.go */

package controller

import (
	"context"
	"fmt"
	"runtime"
	"sync"
	"time"

	"github.com/google/uuid"
)

// TraceSpan represents a single trace span
type TraceSpan struct {
	ID         string
	ParentID   string
	Operation  string
	StartTime  time.Time
	EndTime    time.Time
	Duration   time.Duration
	Metadata   map[string]interface{}
	Tags       map[string]string
	Success    bool
	Error      error
	Component  string
	EntityID   *uint64 // Optional entity ID for monitoring operations
}

// TraceContext holds trace information
type TraceContext struct {
	TraceID string
	SpanID  string
```

```
        Baggage map[string]string
}

// Tracer manages trace collection and storage
type Tracer struct {
        mu        sync.RWMutex
        spans     map[string]*TraceSpan
        traces    map[string][]*TraceSpan // traceID -> spans
        enabled   bool
        component string
        logger    *Logger
}

// NewTracer creates a new tracer instance
func NewTracer(component string, enabled bool) *Tracer {
        // Create a simple logger without tracing to avoid circular dependency
        simpleLogger := &Logger{
                level:       LogLevelDebug,
                component:   fmt.Sprintf("TRACE:%s", component),
                enableColor: false,
                debugMode:   true,
                prodMode:    false,
                timezone:    time.Local,
                tracer:      nil, // No tracer to avoid recursion
        }

        return &Tracer{
                spans:     make(map[string]*TraceSpan),
                traces:    make(map[string][]*TraceSpan),
                enabled:   enabled,
                component: component,
                logger:    simpleLogger,
        }
}

// StartSpan begins a new trace span
func (t *Tracer) StartSpan(ctx context.Context, operation string) (context.Context, *TraceSpan) {
        if !t.enabled {
                return ctx, nil
        }

        var traceID, parentSpanID string

        // Check if there's an existing trace context
        if traceCtx, ok := ctx.Value("traceContext").(*TraceContext); ok {
                traceID = traceCtx.TraceID
                parentSpanID = traceCtx.SpanID
        } else {
                traceID = uuid.New().String()
        }

        spanID := uuid.New().String()

        span := &TraceSpan{
                ID:        spanID,
                ParentID:  parentSpanID,
                Operation: operation,
                StartTime: time.Now(),
                Metadata:  make(map[string]interface{}),
                Tags:      make(map[string]string),
                Component: t.component,
        }

        // Add caller information
        if pc, file, line, ok := runtime.Caller(1); ok {
                span.Tags["caller.file"] = file
                span.Tags["caller.line"] = fmt.Sprintf("%d", line)
                span.Tags["caller.function"] = runtime.FuncForPC(pc).Name()
        }

        t.mu.Lock()
        t.spans[spanID] = span
        t.traces[traceID] = append(t.traces[traceID], span)
        t.mu.Unlock()

        // Create new context with trace information
        newTraceCtx := &TraceContext{
                TraceID: traceID,
                SpanID:  spanID,
                Baggage: make(map[string]string),
```

```go
	}

	newCtx := context.WithValue(ctx, "traceContext", newTraceCtx)

	t.logger.Debug("Started span %s for operation %s (trace: %s, parent: %s)",
		spanID, operation, traceID, parentSpanID)

	return newCtx, span
}

// FinishSpan completes a trace span
func (t *Tracer) FinishSpan(span *TraceSpan, err error) {
	if !t.enabled || span == nil {
		return
	}

	span.EndTime = time.Now()
	span.Duration = span.EndTime.Sub(span.StartTime)
	span.Success = err == nil
	span.Error = err

	t.mu.Lock()
	if existingSpan, exists := t.spans[span.ID]; exists {
		*existingSpan = *span
	}
	t.mu.Unlock()

	status := "SUCCESS"
	if err != nil {
		status = "ERROR"
	}

	t.logger.Debug("Finished span %s (%s) in %v [%s]",
		span.ID, span.Operation, span.Duration, status)

	if err != nil {
		t.logger.Debug("Span %s error: %v", span.ID, err)
	}
}

// AddSpanTag adds a tag to a span
func (t *Tracer) AddSpanTag(span *TraceSpan, key, value string) {
	if !t.enabled || span == nil {
		return
	}
	span.Tags[key] = value
}

// AddSpanMetadata adds metadata to a span
func (t *Tracer) AddSpanMetadata(span *TraceSpan, key string, value interface{}) {
	if !t.enabled || span == nil {
		return
	}
	span.Metadata[key] = value
}

// SetSpanEntity sets the entity ID for a span
func (t *Tracer) SetSpanEntity(span *TraceSpan, entityID uint64) {
	if !t.enabled || span == nil {
		return
	}
	span.EntityID = &entityID
	span.Tags["entity.id"] = fmt.Sprintf("%d", entityID)
}

// GetTrace returns all spans for a trace ID
func (t *Tracer) GetTrace(traceID string) []*TraceSpan {
	if !t.enabled {
		return nil
	}

	t.mu.RLock()
	defer t.mu.RUnlock()

	spans := make([]*TraceSpan, len(t.traces[traceID]))
	copy(spans, t.traces[traceID])
	return spans
}

// GetSpan returns a specific span by ID
```

```go
func (t *Tracer) GetSpan(spanID string) *TraceSpan {
	if !t.enabled {
		return nil
	}

	t.mu.RLock()
	defer t.mu.RUnlock()

	if span, exists := t.spans[spanID]; exists {
		// Return a copy
		spanCopy := *span
		return &spanCopy
	}
	return nil
}

// GetStats returns tracing statistics
func (t *Tracer) GetStats() map[string]interface{} {
	if !t.enabled {
		return map[string]interface{}{"enabled": false}
	}

	t.mu.RLock()
	defer t.mu.RUnlock()

	totalSpans := len(t.spans)
	totalTraces := len(t.traces)

	var avgDuration time.Duration
	var successCount, errorCount int

	if totalSpans > 0 {
		var totalDuration time.Duration
		for _, span := range t.spans {
			if !span.EndTime.IsZero() {
				totalDuration += span.Duration
				if span.Success {
					successCount++
				} else {
					errorCount++
				}
			}
		}
		if totalSpans > 0 {
			avgDuration = totalDuration / time.Duration(totalSpans)
		}
	}

	return map[string]interface{}{
		"enabled":        true,
		"total_spans":    totalSpans,
		"total_traces":   totalTraces,
		"success_count":  successCount,
		"error_count":    errorCount,
		"avg_duration":   avgDuration.String(),
		"component":      t.component,
	}
}

// Cleanup removes old spans to prevent memory leaks
func (t *Tracer) Cleanup(maxAge time.Duration) {
	if !t.enabled {
		return
	}

	t.mu.Lock()
	defer t.mu.Unlock()

	cutoff := time.Now().Add(-maxAge)
	var removedSpans, removedTraces int

	// Remove old spans
	for spanID, span := range t.spans {
		if span.EndTime.Before(cutoff) || (span.EndTime.IsZero() &&
span.StartTime.Before(cutoff)) {
			delete(t.spans, spanID)
			removedSpans++
		}
	}
```

```go
	// Remove empty traces
	for traceID, spans := range t.traces {
		activeSpans := 0
		for _, span := range spans {
			if _, exists := t.spans[span.ID]; exists {
				activeSpans++
			}
		}
		if activeSpans == 0 {
			delete(t.traces, traceID)
			removedTraces++
		}
	}

	if removedSpans > 0 || removedTraces > 0 {
		t.logger.Debug("Cleaned up %d spans and %d traces", removedSpans, removedTraces)
	}
}

// Global tracer instances
var (
	SystemTracer     *Tracer
	SchedulerTracer  *Tracer
	DispatchTracer   *Tracer
	ResultTracer     *Tracer
	WorkerPoolTracer *Tracer
	EntityTracer     *Tracer
)

// InitializeTracers sets up all component tracers
func InitializeTracers(enabled bool) {
	SystemTracer = NewTracer("SYSTEM", enabled)
	SchedulerTracer = NewTracer("SCHEDULER", enabled)
	DispatchTracer = NewTracer("DISPATCH", enabled)
	ResultTracer = NewTracer("RESULT", enabled)
	WorkerPoolTracer = NewTracer("WORKER", enabled)
	EntityTracer = NewTracer("ENTITY", enabled)
}

// StartPeriodicCleanup starts a goroutine that periodically cleans up old traces
func StartPeriodicCleanup(interval, maxAge time.Duration) {
	tracers := []*Tracer{
		SystemTracer, SchedulerTracer, DispatchTracer,
		ResultTracer, WorkerPoolTracer, EntityTracer,
	}

	go func() {
		ticker := time.NewTicker(interval)
		defer ticker.Stop()

		for range ticker.C {
			for _, tracer := range tracers {
				if tracer != nil {
					tracer.Cleanup(maxAge)
				}
			}
		}
	}()
}

/* jobs/jobs.go */

package jobs

import (
	"context"
	"cpra/internal/loader/schema"
	"fmt"
	"github.com/google/uuid"
	"github.com/mlange-42/ark/ecs"
	"github.com/moby/moby/api/types/container"
	"github.com/moby/moby/client"
	"net/http"
	"os"
	"strings"
	"time"
)

// Job defines the interface for any executable task in the system.
type Job interface {
```

```
	Execute() Result
	Copy() Job
	GetEnqueueTime() time.Time
	SetEnqueueTime(time.Time)
	GetStartTime() time.Time
	SetStartTime(time.Time)
}

// CreatePulseJob creates a new pulse job based on the provided schema.
func CreatePulseJob(pulseSchema schema.Pulse, jobID ecs.Entity) (Job, error) {
	timeout := pulseSchema.Timeout
	switch cfg := pulseSchema.Config.(type) {
	case *schema.PulseHTTPConfig:
		return &PulseHTTPJob{
			ID:      uuid.New(),
			Entity:  jobID,
			URL:     strings.Clone(cfg.Url),
			Method:  strings.Clone(cfg.Method),
			Timeout: timeout,
			Retries: cfg.Retries,
			Client:  http.Client{Timeout: timeout},
		}, nil
	case *schema.PulseTCPConfig:
		return &PulseTCPJob{
			ID:      uuid.New(),
			Entity:  jobID,
			Host:    strings.Clone(cfg.Host),
			Port:    cfg.Port,
			Timeout: timeout,
			Retries: cfg.Retries,
		}, nil
	case *schema.PulseICMPConfig:
		return &PulseICMPJob{
			ID:      uuid.New(),
			Entity:  jobID,
			Host:    strings.Clone(cfg.Host),
			Timeout: timeout,
			Count:   cfg.Count,
		}, nil

	// ... other pulse job types
	default:
		return nil, fmt.Errorf("unknown pulse config type: %T for job creation",
pulseSchema.Config)
	}
}

// CreateInterventionJob creates a new intervention job based on the provided schema.
func CreateInterventionJob(interventionSchema schema.Intervention, jobID ecs.Entity) (Job, error) {
	retries := interventionSchema.Retries
	switch interventionSchema.Action {
	case "docker":
		return &InterventionDockerJob{
			ID:        uuid.New(),
			Entity:    jobID,
			Container: strings.Clone(interventionSchema.Target.
(*schema.InterventionTargetDocker).Container),
			Retries:   retries,
			Timeout:   interventionSchema.Target.
(*schema.InterventionTargetDocker).Timeout,
		}, nil
	default:
		return nil, fmt.Errorf("unknown intervention action : %T for job creation",
interventionSchema.Action)
	}
}

// CreateCodeJob creates a new code alert job based on the provided configuration.
func CreateCodeJob(monitor string, config schema.CodeConfig, jobID ecs.Entity, color string) (Job,
error) {
	// ... message creation logic ...
	message := "..."
	switch config.Notify {
	case "log":
		return &CodeLogJob{
			ID:      uuid.New(),
			File:    strings.Clone(config.Config.(*schema.CodeNotificationLog).File),
			Entity:  jobID,
			Monitor: strings.Clone(monitor),
			Message: message,
```

```
			Color:   color,
		}, nil
	case "pagerduty":
		return &CodePagerDutyJob{
			ID:      uuid.New(),
			Entity:  jobID,
			Monitor: strings.Clone(monitor),
			Message: message,
			Color:   color,
		}, nil
	case "slack":
		return &CodeSlackJob{
			ID:      uuid.New(),
			Entity:  jobID,
			Monitor: strings.Clone(monitor),
			Message: message,
			Color:   color,
		}, nil
	case "email":
		return &CodeEmailJob{
			ID:      uuid.New(),
			Entity:  jobID,
			Monitor: strings.Clone(monitor),
			Message: message,
			Color:   color,
		}, nil
	case "webhook":
		return &CodeWebhookJob{
			ID:      uuid.New(),
			Entity:  jobID,
			Monitor: strings.Clone(monitor),
			Message: message,
			Color:   color,
		}, nil
	default:
		return nil, fmt.Errorf("unknown code notification type: %s for job creation",
config.Notify)
	}
}

// --- Pulse Job Implementations ---

type PulseHTTPJob struct {
	ID          uuid.UUID
	Entity      ecs.Entity
	URL         string
	Method      string
	Timeout     time.Duration
	Client      http.Client
	Retries     int
	EnqueueTime time.Time
	StartTime   time.Time
}

func (p *PulseHTTPJob) Execute() Result {
	var lastErr error
	attempts := p.Retries + 1
	payload := map[string]interface{}{"type": "pulse"}

	for i := 0; i < attempts; i++ {
		req, err := http.NewRequest(p.Method, p.URL, nil)
		if err != nil {
			return Result{ID: p.ID, Ent: p.Entity, Err: fmt.Errorf("failed to create http
request: %w", err), Payload: payload}
		}
		resp, err := p.Client.Do(req)
		if err != nil {
			lastErr = err
			time.Sleep(50 * time.Millisecond)
			continue
		}
		defer resp.Body.Close()
		if resp.StatusCode >= 200 && resp.StatusCode < 300 {
			return Result{ID: p.ID, Ent: p.Entity, Err: nil, Payload: payload}
		}
		lastErr = fmt.Errorf("received non-2xx status code: %s", resp.Status)
	}
	return Result{ID: p.ID, Ent: p.Entity, Err: fmt.Errorf("http check failed after %d attempt(s):
%w", attempts, lastErr), Payload: payload}
}
```

```go
func (p *PulseHTTPJob) Copy() Job                  { job := *p; return &job }
func (p *PulseHTTPJob) GetEnqueueTime() time.Time  { return p.EnqueueTime }
func (p *PulseHTTPJob) SetEnqueueTime(t time.Time) { p.EnqueueTime = t }
func (p *PulseHTTPJob) GetStartTime() time.Time    { return p.StartTime }
func (p *PulseHTTPJob) SetStartTime(t time.Time)   { p.StartTime = t }

// PulseTCPJob is a placeholder for a TCP pulse job.
type PulseTCPJob struct {
	ID          uuid.UUID
	Entity      ecs.Entity
	Host        string
	Port        int
	Timeout     time.Duration
	Retries     int
	EnqueueTime time.Time
	StartTime   time.Time
}

func (p *PulseTCPJob) Execute() Result {
	// Mock implementation: does nothing and succeeds.
	return Result{ID: p.ID, Ent: p.Entity, Err: nil, Payload: map[string]interface{}{"type":
"pulse", "driver": "tcp"}}
}

func (p *PulseTCPJob) Copy() Job                  { job := *p; return &job }
func (p *PulseTCPJob) GetEnqueueTime() time.Time  { return p.EnqueueTime }
func (p *PulseTCPJob) SetEnqueueTime(t time.Time) { p.EnqueueTime = t }
func (p *PulseTCPJob) GetStartTime() time.Time    { return p.StartTime }
func (p *PulseTCPJob) SetStartTime(t time.Time)   { p.StartTime = t }

// PulseICMPJob is a placeholder for an ICMP pulse job.
type PulseICMPJob struct {
	ID          uuid.UUID
	Entity      ecs.Entity
	Host        string
	Timeout     time.Duration
	Count       int
	EnqueueTime time.Time
	StartTime   time.Time
}

func (p *PulseICMPJob) Execute() Result {
	// Mock implementation: does nothing and succeeds.
	return Result{ID: p.ID, Ent: p.Entity, Err: nil, Payload: map[string]interface{}{"type":
"pulse", "driver": "icmp"}}
}

func (p *PulseICMPJob) Copy() Job                  { job := *p; return &job }
func (p *PulseICMPJob) GetEnqueueTime() time.Time  { return p.EnqueueTime }
func (p *PulseICMPJob) SetEnqueueTime(t time.Time) { p.EnqueueTime = t }
func (p *PulseICMPJob) GetStartTime() time.Time    { return p.StartTime }
func (p *PulseICMPJob) SetStartTime(t time.Time)   { p.StartTime = t }

// --- Intervention Job Implementations ---

type InterventionDockerJob struct {
	ID          uuid.UUID
	Entity      ecs.Entity
	Container   string
	Timeout     time.Duration
	Retries     int
	EnqueueTime time.Time
	StartTime   time.Time
}

func (i *InterventionDockerJob) Execute() Result {
	payload := map[string]interface{}{"type": "intervention"}
	cli, err := client.NewClientWithOpts(client.FromEnv, client.WithAPIVersionNegotiation())
	if err != nil {
		return Result{ID: i.ID, Ent: i.Entity, Err: fmt.Errorf("failed to create docker
client: %w", err), Payload: payload}
	}
	defer cli.Close()

	var lastErr error
	attempts := i.Retries + 1
	for attempt := 0; attempt < attempts; attempt++ {
		ctx, cancel := context.WithTimeout(context.Background(), i.Timeout)
		defer cancel()
```

```go
                timeout := int(i.Timeout.Seconds())
                restartOptions := container.StopOptions{Timeout: &timeout}
                err := cli.ContainerRestart(ctx, i.Container, restartOptions)
                if err == nil {
                        return Result{ID: i.ID, Ent: i.Entity, Err: nil, Payload: payload}
                }
                lastErr = err
        }
        return Result{ID: i.ID, Ent: i.Entity, Err: fmt.Errorf("docker intervention on '%s' failed
after %d attempt(s): %w", i.Container, attempts, lastErr), Payload: payload}
}

func (i *InterventionDockerJob) Copy() Job                  { job := *i; return &job }
func (i *InterventionDockerJob) GetEnqueueTime() time.Time  { return i.EnqueueTime }
func (i *InterventionDockerJob) SetEnqueueTime(t time.Time) { i.EnqueueTime = t }
func (i *InterventionDockerJob) GetStartTime() time.Time    { return i.StartTime }
func (i *InterventionDockerJob) SetStartTime(t time.Time)   { i.StartTime = t }

// --- Code Job Implementations ---

type CodeLogJob struct {
        ID          uuid.UUID
        Entity      ecs.Entity
        File        string
        Message     string
        Monitor     string
        Color       string
        EnqueueTime time.Time
        StartTime   time.Time
}

func (c *CodeLogJob) Execute() Result {
        payload := map[string]interface{}{"type": "code", "color": c.Color}
        f, err := os.OpenFile(c.File, os.O_APPEND|os.O_CREATE|os.O_WRONLY, 0644)
        if err != nil {
                return Result{ID: c.ID, Ent: c.Entity, Err: err, Payload: payload}
        }
        defer f.Close()

        timestamp := time.Now().Format("2006-01-02 15:04:05.000 Z07:00")
        logLine := fmt.Sprintf("%s [%s] %s\n", timestamp, c.Monitor, c.Message)

        _, err = f.WriteString(logLine)
        return Result{ID: c.ID, Ent: c.Entity, Err: err, Payload: payload}
}

func (c *CodeLogJob) Copy() Job                  { job := *c; return &job }
func (c *CodeLogJob) GetEnqueueTime() time.Time  { return c.EnqueueTime }
func (c *CodeLogJob) SetEnqueueTime(t time.Time) { c.EnqueueTime = t }
func (c *CodeLogJob) GetStartTime() time.Time    { return c.StartTime }
func (c *CodeLogJob) SetStartTime(t time.Time)   { c.StartTime = t }

// CodePagerDutyJob is a placeholder for a PagerDuty notification job.
type CodePagerDutyJob struct {
        ID          uuid.UUID
        Entity      ecs.Entity
        Monitor     string
        Message     string
        Color       string
        EnqueueTime time.Time
        StartTime   time.Time
}

func (c *CodePagerDutyJob) Execute() Result {
        // Mock implementation: does nothing and succeeds.
        return Result{ID: c.ID, Ent: c.Entity, Err: nil, Payload: map[string]interface{}{"type":
"code", "driver": "pagerduty", "color": c.Color}}
}

func (c *CodePagerDutyJob) Copy() Job                  { job := *c; return &job }
func (c *CodePagerDutyJob) GetEnqueueTime() time.Time  { return c.EnqueueTime }
func (c *CodePagerDutyJob) SetEnqueueTime(t time.Time) { c.EnqueueTime = t }
func (c *CodePagerDutyJob) GetStartTime() time.Time    { return c.StartTime }
func (c *CodePagerDutyJob) SetStartTime(t time.Time)   { c.StartTime = t }

// CodeSlackJob is a placeholder for a Slack notification job.
type CodeSlackJob struct {
        ID          uuid.UUID
        Entity      ecs.Entity
        Monitor     string
```

```go
	Message     string
	Color       string
	EnqueueTime time.Time
	StartTime   time.Time
}

func (c *CodeSlackJob) Execute() Result {
	// Mock implementation: does nothing and succeeds.
	return Result{ID: c.ID, Ent: c.Entity, Err: nil, Payload: map[string]interface{}{"type":
"code", "driver": "slack", "color": c.Color}}
}

func (c *CodeSlackJob) Copy() Job                  { job := *c; return &job }
func (c *CodeSlackJob) GetEnqueueTime() time.Time  { return c.EnqueueTime }
func (c *CodeSlackJob) SetEnqueueTime(t time.Time) { c.EnqueueTime = t }
func (c *CodeSlackJob) GetStartTime() time.Time    { return c.StartTime }
func (c *CodeSlackJob) SetStartTime(t time.Time)   { c.StartTime = t }

// CodeEmailJob is a placeholder for an email notification job.
type CodeEmailJob struct {
	ID          uuid.UUID
	Entity      ecs.Entity
	Monitor     string
	Message     string
	Color       string
	EnqueueTime time.Time
	StartTime   time.Time
}

func (c *CodeEmailJob) Execute() Result {
	// Mock implementation: does nothing and succeeds.
	return Result{ID: c.ID, Ent: c.Entity, Err: nil, Payload: map[string]interface{}{"type":
"code", "driver": "email", "color": c.Color}}
}

func (c *CodeEmailJob) Copy() Job                  { job := *c; return &job }
func (c *CodeEmailJob) GetEnqueueTime() time.Time  { return c.EnqueueTime }
func (c *CodeEmailJob) SetEnqueueTime(t time.Time) { c.EnqueueTime = t }
func (c *CodeEmailJob) GetStartTime() time.Time    { return c.StartTime }
func (c *CodeEmailJob) SetStartTime(t time.Time)   { c.StartTime = t }

// CodeWebhookJob is a placeholder for a webhook notification job.
type CodeWebhookJob struct {
	ID          uuid.UUID
	Entity      ecs.Entity
	Monitor     string
	Message     string
	Color       string
	EnqueueTime time.Time
	StartTime   time.Time
}

func (c *CodeWebhookJob) Execute() Result {
	// Mock implementation: does nothing and succeeds.
	return Result{ID: c.ID, Ent: c.Entity, Err: nil, Payload: map[string]interface{}{"type":
"code", "driver": "webhook", "color": c.Color}}
}

func (c *CodeWebhookJob) Copy() Job                  { job := *c; return &job }
func (c *CodeWebhookJob) GetEnqueueTime() time.Time  { return c.EnqueueTime }
func (c *CodeWebhookJob) SetEnqueueTime(t time.Time) { c.EnqueueTime = t }
func (c *CodeWebhookJob) GetStartTime() time.Time    { return c.StartTime }
func (c *CodeWebhookJob) SetStartTime(t time.Time)   { c.StartTime = t }


/* jobs/results.go */

package jobs

import (
	"github.com/google/uuid"
	"github.com/mlange-42/ark/ecs"
)

// Result is a generic structure for returning the outcome of a job.
// It includes the entity it belongs to, any error that occurred, and a flexible payload.
type Result struct {
	ID      uuid.UUID
	Ent     ecs.Entity
	Err     error
```

```go
	Payload map[string]interface{}
}

// Entity returns the entity associated with the result.
func (r *Result) Entity() ecs.Entity {
	return r.Ent
}

// Error returns the error associated with the result, if any.
func (r *Result) Error() error {
	return r.Err
}

/* loader/loader/errors.go */

package loader


/* loader/loader/loader.go */

package loader

import (
	"cpra/internal/loader/schema"
)

type Loader interface {
	Load() error
	GetManifest() schema.Manifest
}

func NewLoader(loaderType string, filename string) Loader {

	switch loaderType {
	case "yaml":
		yamlLoader := NewYamlLoader(filename)
		return yamlLoader

	default:
		yamlLoader := NewYamlLoader(filename)
		return yamlLoader

	}
}


/* loader/loader/loader_test.go */

package loader

import (
	"fmt"
	"testing"
)

func BenchmarkPrimeNumbers(b *testing.B) {
	for i := 0; i < b.N; i++ {
		l := NewYamlLoader("test.yaml")
		l.Load()
		m := l.GetManifest()
		if testing.Verbose() {
			fmt.Printf("loading %d monitors from %s\n", len(m.Monitors), "test.yaml")
		}
	}
}


/* loader/loader/utils.go */

package loader

import (
	"fmt"
	"os"
)

func fatalManifestError(err error) {
	_, err = fmt.Fprintf(os.Stderr, "error: %v\n", err)
	if err != nil {
		return
```

```go
        }
        os.Exit(1)
}


/* loader/loader/yaml_loader.go */

package loader

import (
        "cpra/internal/loader/parser"
        "errors"
        "fmt"
        "strings"

        //"cpra/internal/loader/parser"
        "cpra/internal/loader/schema"
        //"errors"
        //"fmt"
        "gopkg.in/yaml.v3"
        //"strings"

        "os"
)

type YamlLoader struct {
        File     string
        Manifest schema.Manifest
}

func NewYamlLoader(fileName string) *YamlLoader {
        return &YamlLoader{
                fileName,
                schema.Manifest{},
        }
}

func (l *YamlLoader) Load() error {
        file, err := os.Open(l.File)
        defer file.Close()

        if err != nil {
                return err
        }
        //decoder := yaml.NewDecoder(file)
        //var manifest schema.Manifest
        //if err := decoder.Decode(&manifest); err != nil {
        //      // This error will now include line numbers and be very clear
        //      // because it comes directly from the yaml.v3 library.
        //      log.Fatal(err)
        //}
        yamlParser := parser.NewYamlParser()
        manifest, err := yamlParser.Parse(file)
        if err != nil {
                var typeErr *yaml.TypeError
                if errors.As(err, &typeErr) {
                        for _, msg := range typeErr.Errors {
                                if strings.HasPrefix(msg, "line") {
                                        return fmt.Errorf("invalid manifest: %s", msg)
                                }
                        }
                }
                return fmt.Errorf("invalid manifest: %w", err)
        }
        //yamlValidator := validator.NewYamlValidator()
        //err = yamlValidator.ValidateManifest(&manifest)
        //if err != nil {
        //      log.Fatal(err)
        //}
        l.Manifest = manifest
        return nil
}

func (l *YamlLoader) GetManifest() schema.Manifest {
        return l.Manifest
}


/* loader/parser/errors.go */
```

```go
package parser

import "fmt"

var (
	ErrInvalidYamlFormat = fmt.Errorf("invalid yaml format")

	ErrUnknownField     = fmt.Errorf("unknown field")
	ErrInvalidPulseType = fmt.Errorf("invalid pulse type")
	ErrRequiredField    = fmt.Errorf("required field")
	ErrInvalidType      = fmt.Errorf("invalid type")
)

type requiredMonitorFieldError struct {
	monitor   string
	parentKey string
	field     string
	line      int
	reason    error
}

func (e *requiredMonitorFieldError) Error() string {
	if e.monitor == "" {
		return fmt.Sprintf("misssing required %s field %q (line %d)", e.parentKey, e.field,
e.line)
	} else {
		return fmt.Sprintf("misssing required %s field %q in monitor %q (line %d)",
e.parentKey, e.field, e.monitor, e.line)
	}

}

type monitorFieldTypeError struct {
	FieldName string
	FiledType string
	validType string
}

func (r *monitorFieldTypeError) Error() string {
	return fmt.Sprintf("invalid field type %s for %q: valid types are %s", r.FiledType,
r.FieldName, r.validType)
}

type invalidMonitorFieldError struct {
	monitor   string
	parentKey string
	field     string
	line      int
	reason    error
}

func (e *invalidMonitorFieldError) Error() string {
	return fmt.Sprintf("invalid %s field %q in monitor %q (line %d): %s", e.parentKey, e.field,
e.monitor, e.line, e.reason)
}

// Unwrap NOT USED REMOVE LATER
func (e *invalidMonitorFieldError) Unwrap() error {
	return e.reason
}

type duplicateMonitorNameError struct {
	name string
	line int
}

func (e *duplicateMonitorNameError) Error() string {
	return fmt.Sprintf("duplicate monitor name %q (line %d), monitor names must be unique and
cannot be reused", e.name, e.line)
}


/* loader/parser/parser.go */

package parser

import (
	"cpra/internal/loader/schema"
	"io"
)
```

```go
type Parser interface {
	Parse(r io.Reader) (schema.Manifest, error)
}

func NewParser() Parser {
	yamlParser := NewYamlParser()
	return yamlParser
}


/* loader/parser/parser_test.go */

package parser


/* loader/parser/utils.go */

package parser

import (
	"cpra/internal/loader/schema"
	"gopkg.in/yaml.v3"
)

func isValidKey(key string, section string) error {
	f := ManifestFields[section]
	if _, ok := f[key]; !ok {
		return ErrUnknownField
	}
	return nil
}

func checkMissingRequiredKey(section string, node map[string]yaml.Node) (string, error) {
	f := ManifestFields[section]
	for key, field := range f {
		if field.Required {
			if _, ok := node[key]; !ok {
				return key, ErrRequiredField
			}
		}
	}
	return "", nil
}

func decodePulseConfig(config yaml.Node, pulseType string) (schema.PulseConfig, error) {
	switch pulseType {
	case "http":
		var pulseConfig schema.PulseHTTPConfig
		err := config.Decode(&pulseConfig)
		if err != nil {
			return nil, err
		}
		return &pulseConfig, nil
	case "tcp":
		var pulseConfig schema.PulseTCPConfig
		err := config.Decode(&pulseConfig)
		if err != nil {
			return nil, err
		}
		return &pulseConfig, nil
	case "icmp":
		var pulseConfig schema.PulseICMPConfig
		err := config.Decode(&pulseConfig)
		if err != nil {
			return nil, err
		}
		return &pulseConfig, nil
	default:
		return nil, ErrInvalidPulseType
	}
}


/* loader/parser/yaml_parser.go */

package parser

import (
	"cpra/internal/loader/schema"
```

```go
	"fmt"
	"gopkg.in/yaml.v3"
	"io"
)

type FieldType struct {
	Required bool
}

type parseState struct {
	line                      int
	monitorName               string
	pulseType                 string
	interventionTarget        string
	interventionTargetFields string
	codeColor                 string
	fields                    string
}

var (
	MonitorFields = map[string]FieldType{
		"name":          {Required: true},
		"enabled":       {Required: false},
		"pulse_check":   {Required: true},
		"intervention":  {Required: false},
		"codes":         {Required: false},
		"notify_groups": {Required: false},
	}
	PulseFields = map[string]FieldType{
		"type":         {Required: true},
		"interval":     {Required: true},
		"timeout":      {Required: true},
		"max_failures": {Required: false},
		"config":       {Required: true},
	}
	PulseConfigHTTPFields = map[string]FieldType{
		"url":     {Required: true},
		"method":  {Required: false},
		"headers": {Required: false},
		"auth":    {Required: false},
		"retries": {Required: false},
	}

	PulseConfigTCPFields = map[string]FieldType{
		"host":    {Required: true},
		"port":    {Required: true},
		"retries": {Required: false},
	}

	PulseConfigICMPFields = map[string]FieldType{
		"host":             {Required: true},
		"count":            {Required: false},
		"retries":          {Required: false},
		"ignore_privilege": {Required: false},
	}

	// TODO

	//PulseConfigGRPCFields   = map[string]FieldType{}
	//PulseConfigDockerFields = map[string]FieldType{}

	// ????
	//PulseConfigTLSFields = map[string]FieldType{}
	//PulseConfigUDPFields = map[string]FieldType{}
	//PulseConfigDNSFields = map[string]FieldType{}

	InterventionFields = map[string]FieldType{
		"action":       {Required: true},
		"retries":      {Required: false},
		"target":       {Required: true},
		"max_failures": {Required: false},
	}
	InterventionTargetDockerFields = map[string]FieldType{
		"type":      {Required: false},
		"container": {Required: true},
		"timeout":   {Required: false},
	}
	CodeFields = map[string]FieldType{
		"groups": {Required: false},
		"red":    {Required: false},
```

```go
		"yellow": {Required: false},
		"green":  {Required: false},
		"cyan":   {Required: false},
		"gray":   {Required: false},
	}

	CodeColorFields = map[string]FieldType{
		"groups":   {Required: false},
		"dispatch": {Required: false},
		"notify":   {Required: true},
		"config":   {Required: true},
	}
)

var ManifestFields = map[string]map[string]FieldType{
	"monitors":            MonitorFields,
	"pulse_check":         PulseFields,
	"pulse_check_http":    PulseConfigHTTPFields,
	"pulse_check_tcp":     PulseConfigTCPFields,
	"pulse_check_icmp":    PulseConfigICMPFields,
	"intervention":        InterventionFields,
	"intervention_docker": InterventionTargetDockerFields,
	"codes":               CodeFields,
	"code_color":          CodeColorFields,
}

type YamlParser struct {
}

func NewYamlParser() *YamlParser {
	return &YamlParser{}
}

func (p *YamlParser) Parse(r io.Reader) (schema.Manifest, error) {

	var state parseState
	var manifest schema.Manifest
	decoder := yaml.NewDecoder(r)

	for {
		var node map[string]yaml.Node
		err := decoder.Decode(&node)
		if err == io.EOF {
			break
		}
		if err != nil {
			return schema.Manifest{}, err
		}
		for key, value := range node {
			if key == "monitors" {
				var monitorsNode []yaml.Node
				if err := value.Decode(&monitorsNode); err != nil {
					return schema.Manifest{}, err
				}
				seen := map[string]struct{}{}

				for _, monitor := range monitorsNode {

					m, err := p.ParseMonitor(monitor, &state)
					if err != nil {
						return schema.Manifest{}, err
					}
					name := state.monitorName
					if _, exists := seen[name]; exists {
						return schema.Manifest{}, &duplicateMonitorNameError{
							name: name,
							line: monitor.Content[0].Line,
						}
					}
					seen[state.monitorName] = struct{}{}

					manifest.Monitors = append(manifest.Monitors, m)
				}
			}
		}

	}

	return manifest, nil
}
```

```
func (p *YamlParser) ParseMonitor(m yaml.Node, state *parseState) (schema.Monitor, error) {
	var keys map[string]yaml.Node
	monitor := schema.Monitor{}
	if err := m.Decode(&keys); err != nil {
		return schema.Monitor{}, err
	}
	node, ok := keys["name"]
	if !ok {
		node := m.Content[0]
		return schema.Monitor{}, &requiredMonitorFieldError{
			field:     "name",
			parentKey: "monitor",
			line:      node.Line,
			reason:    ErrRequiredField,
		}
	}
	state.monitorName = node.Value
	state.line = node.Line
	key, err := checkMissingRequiredKey("monitors", keys)

	if err != nil || key != "" {
		return schema.Monitor{}, &requiredMonitorFieldError{
			field:     key,
			parentKey: "monitor",
			line:      m.Line,
			reason:    ErrRequiredField,
		}
	}

	for k := range keys {
		err := isValidKey(k, "monitors")
		if err != nil {
			line := keys[k].Line

			return schema.Monitor{}, &invalidMonitorFieldError{parentKey: "monitor",
monitor: state.monitorName, field: k, line: line, reason: err}
		}

	}
	monitor.Name = state.monitorName
	if enabled, ok := keys["enabled"]; ok {
		if enabled.Value == "false" {
			monitor.Enabled = false
		} else {
			monitor.Enabled = true
		}
	} else {
		monitor.Enabled = true
	}

	pulseNode := keys["pulse_check"]

	pulse, err := p.ParsePulse(pulseNode, state)

	if err != nil {
		return schema.Monitor{}, err
	}

	monitor.Pulse = pulse

	if interventionNode, ok := keys["intervention"]; ok {
		intervention, err := p.ParseIntervention(interventionNode, state)
		if err != nil {
			return schema.Monitor{}, err
		}
		monitor.Intervention = intervention
	}

	codeNode := keys["codes"]
	codes, err := p.ParseCode(codeNode, state)
	if err != nil {
		return schema.Monitor{}, err
	}
	monitor.Codes = codes

	return monitor, nil
}

func (p *YamlParser) ParsePulse(pNode yaml.Node, state *parseState) (schema.Pulse, error) {
```

```
	var keys map[string]yaml.Node

	if err := pNode.Decode(&keys); err != nil {
		return schema.Pulse{}, err
	}

	key, err := checkMissingRequiredKey("pulse_check", keys)

	if err != nil || key != "" {
		return schema.Pulse{}, &requiredMonitorFieldError{
			field:     key,
			parentKey: "pulse_check",
			line:      pNode.Line,
			reason:    ErrRequiredField,
		}
	}

	for k := range keys {
		err := isValidKey(k, "pulse_check")
		if err != nil {
			line := keys[k].Line

			return schema.Pulse{}, &invalidMonitorFieldError{parentKey: "pulse_check", 
monitor: state.monitorName, field: k, line: line, reason: err}
		}
	}

	state.pulseType = keys["type"].Value

	switch state.pulseType {
	case "http":
		state.fields = "pulse_check_http"
	case "tcp":
		state.fields = "pulse_check_tcp"
	case "icmp":
		state.fields = "pulse_check_icmp"

	default:
		return schema.Pulse{}, &invalidMonitorFieldError{parentKey: "pulse_check", monitor: 
state.monitorName, field: "type", line: keys["type"].Line, reason: ErrUnknownField}
	}

	state.line = pNode.Content[0].Line
	pConfig := keys["config"]
	config, err := p.ParsePulseConfig(pConfig, state)
	if err != nil {
		return schema.Pulse{}, err
	}

	var pulse schema.Pulse
	err = pNode.Decode(&pulse)
	if err != nil {
		return schema.Pulse{}, err
	}
	pulse.Config = config
	return pulse, nil
}

func (p *YamlParser) ParsePulseConfig(pNode yaml.Node, state *parseState) (schema.PulseConfig, error) 
{
	var keys map[string]yaml.Node
	if err := pNode.Decode(&keys); err != nil {
		return nil, err
	}
	state.line = pNode.Content[0].Line

	key, err := checkMissingRequiredKey(state.fields, keys)

	if err != nil || key != "" {
		return nil, &requiredMonitorFieldError{
			field:     key,
			parentKey: fmt.Sprintf("%v pulse_check", state.pulseType),
			line:      state.line,
			reason:    ErrRequiredField,
		}
	}

	for k := range keys {
		err := isValidKey(k, state.fields)
		if err != nil {
```

```
                        line := keys[k].Line
                        return nil, &invalidMonitorFieldError{parentKey: "pulse_check", monitor:
state.monitorName, field: k, line: line, reason: fmt.Errorf("invalid pulse config for type %q %w",
keys["type"].Value, err)}
                }
        }

        pulseType, err := decodePulseConfig(pNode, state.pulseType)

        if err != nil {
                return nil, err
        }

        return pulseType, nil
}

func (p *YamlParser) ParseIntervention(i yaml.Node, state *parseState) (schema.Intervention, error) {
        var keys map[string]yaml.Node
        if err := i.Decode(&keys); err != nil {
                return schema.Intervention{}, err
        }

        key, err := checkMissingRequiredKey("intervention", keys)

        if err != nil || key != "" {
                return schema.Intervention{}, &requiredMonitorFieldError{
                        field:     key,
                        parentKey: "intervention",
                        line:      i.Line,
                        reason:    ErrRequiredField,
                }
        }

        for k := range keys {
                err := isValidKey(k, "intervention")
                if err != nil {
                        line := keys[k].Line

                        return schema.Intervention{}, &invalidMonitorFieldError{parentKey:
"intervention", monitor: state.monitorName, field: k, line: line, reason: err}
                }
        }
        iTarget := keys["target"]

        var target map[string]yaml.Node
        if err := iTarget.Decode(&target); err != nil {
                return schema.Intervention{}, err
        }

        //targetType, ok := target["type"]
        //if !ok {
        //      return schema.Intervention{}, &requiredMonitorFieldError{
        //              field:     "type",
        //              parentKey: "intervention.target",
        //              line:      targetType.Line,
        //              reason:    ErrRequiredField,
        //      }
        //}
        action := keys["action"]
        switch action.Value {
        case "docker":
                state.interventionTargetFields = "intervention_docker"
                err := p.ParseInterventionTarget(iTarget, state)
                if err != nil {
                        return schema.Intervention{}, err
                }
        default:
                return schema.Intervention{}, &invalidMonitorFieldError{parentKey: "intervention",
monitor: state.monitorName, field: "target", line: keys["target"].Content[0].Line, reason:
ErrUnknownField}
        }
        var intervention schema.Intervention
        err = i.Decode(&intervention)
        if err != nil {
                return schema.Intervention{}, err
        }
        //pulse.Config = config
        return intervention, nil
}
```

```
func (p *YamlParser) ParseInterventionTarget(i yaml.Node, state *parseState) error {
	var keys map[string]yaml.Node

	if err := i.Decode(&keys); err != nil {
		return err
	}

	key, err := checkMissingRequiredKey(state.interventionTargetFields, keys)

	if err != nil || key != "" {
		return &requiredMonitorFieldError{
			field:     key,
			parentKey: fmt.Sprintf("%v intervention", keys["type"].Value),
			line:      i.Line,
			reason:    ErrRequiredField,
		}
	}

	for k := range keys {
		err := isValidKey(k, state.interventionTargetFields)
		if err != nil {
			line := keys[k].Line
			return &invalidMonitorFieldError{parentKey: fmt.Sprintf("%v intervention",
keys["type"].Value), monitor: state.monitorName, field: k, line: line, reason: err}

		}
	}
	return nil
}

func (p *YamlParser) ParseCode(c yaml.Node, state *parseState) (schema.Codes, error) {
	var keys map[string]yaml.Node
	if err := c.Decode(&keys); err != nil {
		return nil, err
	}
	var codes schema.Codes
	if err := c.Decode(&codes); err != nil {
		return nil, err
	}
	key, err := checkMissingRequiredKey("codes", keys)
	if err != nil || key != "" {
		return nil, &requiredMonitorFieldError{
			field:     key,
			parentKey: "codes",
			line:      c.Line,
			reason:    ErrRequiredField,
		}
	}

	//codes := make(schema.Codes)

	for k, _ := range keys {
		err := isValidKey(k, "codes")
		if err != nil {
			line := keys[k].Line
			return nil, &invalidMonitorFieldError{parentKey: "codes", monitor:
state.monitorName, field: k, line: line, reason: err}
		}

	}
	for colorKey, colorNode := range keys {
		// ... (existing key validation)

		// ADD THIS LOGIC
		state.codeColor = colorKey // Set the state for the color being parsed
		err := p.ParseCodeColor(colorNode, state)
		if err != nil {
			return nil, err
		}
	}
	return codes, nil
}

func (p *YamlParser) ParseCodeColor(c yaml.Node, state *parseState) error {
	var keys map[string]yaml.Node
	if err := c.Decode(&keys); err != nil {
		return err
	}

	key, err := checkMissingRequiredKey("code_color", keys)
```

```go
	if err != nil || key != "" {
		return &requiredMonitorFieldError{
			field:     key,
			parentKey: state.codeColor,
			line:      c.Line,
			reason:    ErrRequiredField,
		}
	}

	for k := range keys {
		err := isValidKey(k, "code_color")
		if err != nil {
			line := keys[k].Line
			return &invalidMonitorFieldError{parentKey: state.codeColor, monitor:
state.monitorName, field: k, line: line, reason: err}
		}
	}
	return nil
}


/* loader/schema/manifest.go */

package schema

import (
	"encoding/json"
	"fmt"
	"strings"
	"time"

	"gopkg.in/yaml.v3"
)

//// UTILITY TYPES

type DurationSeconds int

func (d *DurationSeconds) UnmarshalYAML(unmarshal func(interface{}) error) error {
	var raw string
	if err := unmarshal(&raw); err != nil {
		return fmt.Errorf("invalid duration %q", raw)
	}
	p, err := time.ParseDuration(raw)
	if err != nil {
		return fmt.Errorf("invalid duration %q: %w", raw, err)
	}
	*d = DurationSeconds(int(p.Seconds()))
	return nil
}

type StringList []string

func (s *StringList) UnmarshalYAML(unmarshal func(interface{}) error) error {
	var single string
	if err := unmarshal(&single); err == nil {
		*s = []string{single}
		return nil
	}
	var multi []string
	if err := unmarshal(&multi); err == nil {
		*s = multi
		return nil
	}
	return fmt.Errorf("value must be a string or list of strings")
}

//// PULSE TYPES

type PulseConfig interface {
	isPulseConfigs()
	Copy() PulseConfig
}

type PulseHTTPConfig struct {
	Url     string     `yaml:"url" json:"url"`
	Method  string     `yaml:"method" json:"method"`
	Headers StringList `yaml:"headers" json:"headers"`
	Retries int        `yaml:"retries" json:"retries"`
```

```go
}

func (c *PulseHTTPConfig) Copy() PulseConfig {
	// This was already correct, but for consistency, we'll return a pointer
	// to a new struct.

	newConfig := new(PulseHTTPConfig)
	*newConfig = *c
	return newConfig
}

func (*PulseHTTPConfig) isPulseConfigs() {}

type PulseTCPConfig struct {
	Host    string `yaml:"host"`
	Port    int    `yaml:"port"`
	Retries int    `yaml:"retries"`
}

func (c *PulseTCPConfig) Copy() PulseConfig {
	newConfig := new(PulseTCPConfig)
	*newConfig = *c
	return newConfig
}

func (*PulseTCPConfig) isPulseConfigs() {}

type PulseICMPConfig struct {
	Host      string `yaml:"host"`
	Privilege bool   `yaml:"ignore_privilege"`
	Count     int    `yaml:"count"`
	Retries   int    `yaml:"retries"`
}

func (c *PulseICMPConfig) Copy() PulseConfig {
	newConfig := new(PulseICMPConfig)
	*newConfig = *c
	return newConfig
}

func (*PulseICMPConfig) isPulseConfigs() {}

type Pulse struct {
	Type        string        `yaml:"type" json:"type"`
	Interval    time.Duration `yaml:"interval" json:"interval"`
	Timeout     time.Duration `yaml:"timeout" json:"timeout"`
	MaxFailures int           `yaml:"max_failures" json:"max_failures"`
	Groups      StringList    `yaml:"groups" json:"groups"`
	Config      PulseConfig   `json:"config"`
}

type rawPulse struct {
	Type        string        `yaml:"type"`
	Interval    time.Duration `yaml:"interval"`
	Timeout     time.Duration `yaml:"timeout"`
	Retries     int           `yaml:"retries"`
	MaxFailures int           `yaml:"max_failures"`
	Groups      StringList    `yaml:"groups"`
}

func (p *Pulse) UnmarshalYAML(value *yaml.Node) error {
	var temp struct {
		Config   yaml.Node `yaml:"config"`
		rawPulse `yaml:",inline"`
	}
	if err := value.Decode(&temp); err != nil {
		return err
	}
	*p = Pulse{
		Type:        temp.Type,
		Interval:    temp.Interval,
		Timeout:     temp.Timeout,
		MaxFailures: temp.MaxFailures,
		Groups:      temp.Groups,
	}
	switch temp.Type {
	case "http":
		var c = &PulseHTTPConfig{} // FIX: Allocate on the heap
		if err := temp.Config.Decode(c); err != nil {
			return err
```

```go
		}
		p.Config = c
	case "tcp":
		var c = &PulseTCPConfig{} // FIX: Allocate on the heap
		if err := temp.Config.Decode(c); err != nil {
			return err
		}
		p.Config = c
	case "icmp":
		var c = &PulseICMPConfig{} // FIX: Allocate on the heap
		if err := temp.Config.Decode(c); err != nil {
			return err
		}
		p.Config = c
	default:
		return fmt.Errorf("unknown pulse type: %q", temp.Type)
	}
	return nil
}

// UnmarshalJSON handles JSON unmarshaling for Pulse (needed for JSON parser)
func (p *Pulse) UnmarshalJSON(data []byte) error {
	var temp struct {
		Type        string          `json:"type"`
		Interval    string          `json:"interval"`    // Parse as string first
		Timeout     string          `json:"timeout"`     // Parse as string first
		MaxFailures int             `json:"max_failures"`
		Config      json.RawMessage `json:"config"`
	}

	if err := json.Unmarshal(data, &temp); err != nil {
		return err
	}

	// Parse duration strings
	interval, err := time.ParseDuration(temp.Interval)
	if err != nil {
		return fmt.Errorf("invalid interval duration %q: %w", temp.Interval, err)
	}

	timeout, err := time.ParseDuration(temp.Timeout)
	if err != nil {
		return fmt.Errorf("invalid timeout duration %q: %w", temp.Timeout, err)
	}

	*p = Pulse{
		Type:        temp.Type,
		Interval:    interval,
		Timeout:     timeout,
		MaxFailures: temp.MaxFailures,
	}

	switch temp.Type {
	case "http":
		var c = &PulseHTTPConfig{}
		if err := json.Unmarshal(temp.Config, c); err != nil {
			return err
		}
		p.Config = c
	case "tcp":
		var c = &PulseTCPConfig{}
		if err := json.Unmarshal(temp.Config, c); err != nil {
			return err
		}
		p.Config = c
	case "icmp":
		var c = &PulseICMPConfig{}
		if err := json.Unmarshal(temp.Config, c); err != nil {
			return err
		}
		p.Config = c
	default:
		return fmt.Errorf("unknown pulse type: %q", temp.Type)
	}
	return nil
}

//// INTERVENTION TYPES

type Intervention struct {
```

```go
	Action      string             `yaml:"action"`
	Retries     int                `yaml:"retries"`
	Target      InterventionTarget `yaml:"target"`
	MaxFailures int                `yaml:"max_failures"`
}
type rawIntervention struct {
	Action  string `yaml:"action"`
	Retries int    `yaml:"retries"`
}

func (i *Intervention) UnmarshalYAML(value *yaml.Node) error {
	var temp struct {
		Target          yaml.Node `yaml:"target"`
		rawIntervention `yaml:",inline"`
	}
	if err := value.Decode(&temp); err != nil {
		return err
	}
	*i = Intervention{
		Action:  temp.Action,
		Retries: temp.Retries,
	}
	switch temp.Action {
	case "docker":
		var t = &InterventionTargetDocker{} // FIX: Allocate on the heap
		if err := temp.Target.Decode(t); err != nil {
			return err
		}
		i.Target = t
	default:
		return fmt.Errorf("unknown intervention type: %q", temp.Action)
	}
	return nil
}

// UnmarshalJSON handles JSON unmarshaling for Intervention (needed for JSON parser)
func (i *Intervention) UnmarshalJSON(data []byte) error {
	var temp struct {
		Action  string          `json:"action"`
		Retries int             `json:"retries"`
		Target  json.RawMessage `json:"target"`
	}

	if err := json.Unmarshal(data, &temp); err != nil {
		return err
	}

	*i = Intervention{
		Action:  temp.Action,
		Retries: temp.Retries,
	}

	switch temp.Action {
	case "docker":
		var t = &InterventionTargetDocker{}
		if err := json.Unmarshal(temp.Target, t); err != nil {
			return err
		}
		i.Target = t
	default:
		return fmt.Errorf("unknown intervention type: %q", temp.Action)
	}
	return nil
}

type InterventionTarget interface {
	GetTargetType() string
	Copy() InterventionTarget
}

type InterventionTargetDocker struct {
	Type      string        `yaml:"type" json:"type"`
	Container string        `yaml:"container" json:"container"`
	Timeout   time.Duration `yaml:"timeout" json:"timeout"`
}

func (i *InterventionTargetDocker) Copy() InterventionTarget {
	return &InterventionTargetDocker{
		Type:      strings.Clone(i.Type),
		Container: strings.Clone(i.Container),
```

```go
	}
}

func (i *InterventionTargetDocker) GetTargetType() string {
	return i.Type
}

type CodeNotification interface {
	IsCodeNotification()
	Copy() CodeNotification
}

type CodeNotificationLog struct {
	File string `yaml:"file" json:"file"`
}

func (c *CodeNotificationLog) Copy() CodeNotification {
	return &CodeNotificationLog{
		File: strings.Clone(c.File),
	}
}

func (c *CodeNotificationLog) IsCodeNotification() {
}

type CodeNotificationPagerDuty struct {
	URL string `yaml:"url" json:"url"`
}

func (c *CodeNotificationPagerDuty) Copy() CodeNotification {
	return &CodeNotificationPagerDuty{
		URL: strings.Clone(c.URL),
	}
}

func (c *CodeNotificationPagerDuty) IsCodeNotification() {
}

type CodeNotificationSlack struct {
	WebHook string `yaml:"hook" json:"hook"`
}

func (c *CodeNotificationSlack) Copy() CodeNotification {
	return &CodeNotificationSlack{
		WebHook: strings.Clone(c.WebHook),
	}
}

func (c *CodeNotificationSlack) IsCodeNotification() {
}

type CodeConfig struct {
	Dispatch bool             `yaml:"dispatch"`
	Notify   string           `yaml:"notify"`
	Config   CodeNotification `yaml:"config"` // or more specific struct if desired
}

type Codes map[string]CodeConfig

type rawCodes struct {
	Dispatch bool   `yaml:"dispatch"`
	Notify   string `yaml:"notify"`
}

func (c *Codes) UnmarshalYAML(value *yaml.Node) error {
	var codes map[string]yaml.Node
	if err := value.Decode(&codes); err != nil {
		return err
	}
	colors := make(map[string]CodeConfig)
	for color, config := range codes {
		var temp struct {
			Config   yaml.Node `yaml:"config"`
			rawCodes `yaml:",inline"`
		}
		if err := config.Decode(&temp); err != nil {
			return err
		}
		switch temp.Notify {
		case "log":
```

```go
			var t = &CodeNotificationLog{} // FIX: Allocate on the heap
			if err := temp.Config.Decode(t); err != nil {
				return err
			}
			colors[color] = CodeConfig{
				Dispatch: temp.Dispatch,
				Notify:   temp.Notify,
				Config:   t,
			}
		case "slack":
			var t = &CodeNotificationSlack{} // FIX: Allocate on the heap
			if err := temp.Config.Decode(t); err != nil {
				return err
			}
			colors[color] = CodeConfig{
				Dispatch: temp.Dispatch,
				Notify:   temp.Notify,
				Config:   t,
			}
		case "pagerduty":
			var t = &CodeNotificationPagerDuty{} // FIX: Allocate on the heap
			if err := temp.Config.Decode(t); err != nil {
				return err
			}
			colors[color] = CodeConfig{
				Dispatch: temp.Dispatch,
				Notify:   temp.Notify,
				Config:   t,
			}
		default:
			return fmt.Errorf("unknown notificiation type: %q", temp.Notify)
		}
	}
	*c = colors
	return nil
}

// UnmarshalJSON handles JSON unmarshaling for Codes (needed for JSON parser)
func (c *Codes) UnmarshalJSON(data []byte) error {
	var codes map[string]struct {
		Dispatch bool            `json:"dispatch"`
		Notify   string          `json:"notify"`
		Config   json.RawMessage `json:"config"`
	}

	if err := json.Unmarshal(data, &codes); err != nil {
		return err
	}

	colors := make(map[string]CodeConfig)
	for color, config := range codes {
		switch config.Notify {
		case "log":
			var t = &CodeNotificationLog{}
			if err := json.Unmarshal(config.Config, t); err != nil {
				return err
			}
			colors[color] = CodeConfig{
				Dispatch: config.Dispatch,
				Notify:   config.Notify,
				Config:   t,
			}
		case "slack":
			var t = &CodeNotificationSlack{}
			if err := json.Unmarshal(config.Config, t); err != nil {
				return err
			}
			colors[color] = CodeConfig{
				Dispatch: config.Dispatch,
				Notify:   config.Notify,
				Config:   t,
			}
		case "pagerduty":
			var t = &CodeNotificationPagerDuty{}
			if err := json.Unmarshal(config.Config, t); err != nil {
				return err
			}
			colors[color] = CodeConfig{
				Dispatch: config.Dispatch,
				Notify:   config.Notify,
```

```go
                                Config:   t,
                        }
                default:
                        return fmt.Errorf("unknown notification type: %q", config.Notify)
                }
        }
        *c = colors
        return nil
}

type Monitor struct {
        Name         string       `yaml:"name" json:"name"`
        Enabled      bool         `yaml:"enabled" json:"enabled"`
        Pulse        Pulse        `yaml:"pulse_check" json:"pulse_check"`
        Intervention Intervention `yaml:"intervention,omitempty" json:"intervention,omitempty"`
        Codes        Codes        `yaml:"codes" json:"codes"`
}

// UnmarshalYAML sets default values for the Monitor struct, specifically for the Enabled field.
func (m *Monitor) UnmarshalYAML(value *yaml.Node) error {
        // Create a temporary struct with a pointer to a bool for 'Enabled'
        type TmpMonitor struct {
                Name         string       `yaml:"name"`
                Enabled      *bool        `yaml:"enabled"`
                Pulse        Pulse        `yaml:"pulse_check"`
                Intervention Intervention `yaml:"intervention,omitempty"`
                Codes        Codes        `yaml:"codes"`
        }

        var tmp TmpMonitor
        if err := value.Decode(&tmp); err != nil {
                return err
        }

        // Assign fields to the actual monitor struct
        m.Name = tmp.Name
        m.Pulse = tmp.Pulse
        m.Intervention = tmp.Intervention
        m.Codes = tmp.Codes

        // Set 'Enabled' to true if it's not specified in the YAML
        if tmp.Enabled == nil {
                m.Enabled = true
        } else {
                m.Enabled = *tmp.Enabled
        }

        return nil
}

type Manifest struct {
        Monitors []Monitor `yaml:"monitors" json:"monitors"`
}


/* loader/streaming/streaming_entity_creator.go */

package streaming

import (
        "context"
        "cpra/internal/controller/entities"
        "fmt"
        "sync"
        "time"

        "github.com/mlange-42/ark/ecs"
)

// StreamingEntityCreator handles batch entity creation for Ark ECS.
// It now uses the consolidated EntityManager to create entities.
type StreamingEntityCreator struct {
        world         *ecs.World
        entityManager *entities.EntityManager

        // Statistics
        entitiesCreated  int64
        batchesProcessed int64
        startTime        time.Time
        pulseRate        float64
```

```go
	mu                sync.RWMutex
}

// EntityCreationConfig holds entity creation configuration.
type EntityCreationConfig struct {
	BatchSize    int
	PreAllocate  int
	ProgressChan chan<- EntityProgress
}

// EntityProgress represents entity creation progress.
type EntityProgress struct {
	EntitiesCreated  int64
	BatchesProcessed int64
	Rate             float64
	MemoryUsage      int64
}

// NewStreamingEntityCreator creates a new, simplified entity creator.
func NewStreamingEntityCreator(world *ecs.World, config EntityCreationConfig) *StreamingEntityCreator
{
	creator := &StreamingEntityCreator{
		world:         world,
		entityManager: entities.NewEntityManager(world),
		startTime:     time.Now(),
	}

	if config.PreAllocate > 0 {
		creator.preAllocateEntities(config.PreAllocate)
	}

	return creator
}

// preAllocateEntities pre-allocates entity storage to reduce allocations during creation.
func (c *StreamingEntityCreator) preAllocateEntities(count int) {
	tempEntities := make([]ecs.Entity, count)
	for i := 0; i < count; i++ {
		tempEntities[i] = c.world.NewEntity()
	}
	for _, entity := range tempEntities {
		c.world.RemoveEntity(entity)
	}
}

// ProcessBatches processes monitor batches and creates entities.
func (c *StreamingEntityCreator) ProcessBatches(ctx context.Context, batchChan <-chan MonitorBatch,
progressChan chan<- EntityProgress) error {
	progressTicker := time.NewTicker(2 * time.Second)
	defer progressTicker.Stop()

	for {
		select {
		case <-ctx.Done():
			return ctx.Err()

		case <-progressTicker.C:
			c.reportProgress(progressChan)

		case batch, ok := <-batchChan:
			if !ok {
				c.reportProgress(progressChan)
				return nil
			}

			if err := c.processBatch(batch); err != nil {
				return fmt.Errorf("failed to process batch %d: %w", batch.BatchID,
err)
			}
		}
	}
}

// processBatch creates entities for a single batch of monitors.
func (c *StreamingEntityCreator) processBatch(batch MonitorBatch) error {
	var pulseSum float64
	for _, monitor := range batch.Monitors {
		if err := c.entityManager.CreateEntityFromMonitor(&monitor, c.world); err != nil {
			return fmt.Errorf("failed to create entity for monitor '%s': %w",
monitor.Name, err)
```

```
		}
		if monitor.Enabled && monitor.Pulse.Interval > 0 {
			sec := monitor.Pulse.Interval.Seconds()
			if sec > 0 {
				pulseSum += 1.0 / sec
			}
		}
	}

	c.mu.Lock()
	c.entitiesCreated += int64(len(batch.Monitors))
	c.batchesProcessed++
	c.pulseRate += pulseSum
	c.mu.Unlock()

	return nil
}

// reportProgress sends a progress update.
func (c *StreamingEntityCreator) reportProgress(progressChan chan<- EntityProgress) {
	if progressChan == nil {
		return
	}

	c.mu.RLock()
	entitiesCreated := c.entitiesCreated
	batchesProcessed := c.batchesProcessed
	c.mu.RUnlock()

	elapsed := time.Since(c.startTime)
	rate := 0.0
	if elapsed.Seconds() > 0 {
		rate = float64(entitiesCreated) / elapsed.Seconds()
	}

	select {
	case progressChan <- EntityProgress{
		EntitiesCreated:  entitiesCreated,
		BatchesProcessed: batchesProcessed,
		Rate:             rate,
	}:
	default:
	}
}

// PulseRate returns the aggregated expected pulse arrival rate (jobs/sec).
func (c *StreamingEntityCreator) PulseRate() float64 {
	c.mu.RLock()
	defer c.mu.RUnlock()
	return c.pulseRate
}

// GetStats returns current creation statistics.
func (c *StreamingEntityCreator) GetStats() (entitiesCreated int64, batchesProcessed int64, rate
float64) {
	c.mu.RLock()
	defer c.mu.RUnlock()

	elapsed := time.Since(c.startTime)
	if elapsed.Seconds() == 0 {
		return c.entitiesCreated, c.batchesProcessed, 0
	}
	return c.entitiesCreated, c.batchesProcessed, float64(c.entitiesCreated) / elapsed.Seconds()
}


/* loader/streaming/streaming_json_parser.go */

package streaming

import (
	"context"
	"encoding/json"
	"fmt"
	"io"
	"os"

	"cpra/internal/loader/schema"
)
```

```go
// StreamingJsonParser handles true streaming parsing of a JSON file.
// It reads the file object by object, creating batches without loading the entire file into memory.
type StreamingJsonParser struct {
	filename string
	config   ParseConfig
}

// NewStreamingJsonParser creates a new streaming JSON parser.
func NewStreamingJsonParser(filename string, config ParseConfig) (*StreamingJsonParser, error) {
	return &StreamingJsonParser{
		filename: filename,
		config:   config,
	}, nil
}

// ParseBatches streams the JSON file and sends batches of monitors over a channel.
func (p *StreamingJsonParser) ParseBatches(ctx context.Context, progressChan chan<- Progress) (<-chan
MonitorBatch, <-chan error) {
	batchChan := make(chan MonitorBatch, 100) // Buffer for a few batches
	errorChan := make(chan error, 1)

	go func() {
		defer close(batchChan)
		defer close(errorChan)

		if err := p.parseFile(ctx, batchChan, progressChan); err != nil {
			errorChan <- err
		}
	}()

	return batchChan, errorChan
}

// parseFile performs the actual streaming JSON parsing.
func (p *StreamingJsonParser) parseFile(ctx context.Context, batchChan chan<- MonitorBatch,
progressChan chan<- Progress) error {
	file, err := os.Open(p.filename)
	if err != nil {
		return fmt.Errorf("failed to open file: %w", err)
	}
	defer file.Close()

	decoder := json.NewDecoder(file)

	// Read the opening bracket of the object.
	t, err := decoder.Token()
	if err != nil {
		return fmt.Errorf("failed to read opening token: %w", err)
	}
	if t != json.Delim('{') {
		return fmt.Errorf("expected { at start of json file, got %v", t)
	}

	// Find the "monitors" key
	for decoder.More() {
		t, err := decoder.Token()
		if err != nil {
			return fmt.Errorf("failed to read token: %w", err)
		}
		if s, ok := t.(string); ok && s == "monitors" {
			break
		}
	}

	// Read the opening bracket of the array.
	t, err = decoder.Token()
	if err != nil {
		return fmt.Errorf("failed to read opening token: %w", err)
	}
	if t != json.Delim('[') {
		return fmt.Errorf("expected [ after 'monitors' key, got %v", t)
	}

	batchID := 0
	// Loop while there are more objects in the array.
	for decoder.More() {
		select {
		case <-ctx.Done():
			return ctx.Err()
		default:
```

```go
			batch := make([]schema.Monitor, 0, p.config.BatchSize)
			// Fill a batch
			for i := 0; i < p.config.BatchSize && decoder.More(); i++ {
				var monitor schema.Monitor
				if err := decoder.Decode(&monitor); err != nil {
					return fmt.Errorf("failed to decode monitor object: %w", err)
				}
				batch = append(batch, monitor)
			}

			// Send the batch to the creator
			batchChan <- MonitorBatch{
				Monitors: batch,
				BatchID:  batchID,
			}
			batchID++
		}
	}

	// Read the closing bracket of the array.
	t, err = decoder.Token()
	if err != nil && err != io.EOF {
		return fmt.Errorf("failed to read closing token: %w", err)
	}
	if t != json.Delim(']') {
		return fmt.Errorf("expected ] at end of json file, got %v", t)
	}

	return nil
}


/* loader/streaming/streaming_loader.go */

package streaming

import (
	"context"
	"cpra/internal/loader/schema"
	"fmt"
	"runtime"
	"strings"
	"time"

	"github.com/mlange-42/ark/ecs"
)

// MonitorBatch represents a batch of monitors read from a file.
type MonitorBatch struct {
	Monitors []schema.Monitor
	BatchID  int
	Offset   int64
}

// ParseConfig holds configuration for the streaming parsers.
type ParseConfig struct {
	BatchSize    int
	BufferSize   int
	MaxMemory    int64
	ProgressChan chan<- Progress
}

// Progress represents parsing progress.
type Progress struct {
	EntitiesProcessed  int64
	TotalBytes         int64
	ProcessedBytes     int64
	Percentage         float64
	Rate               float64 // entities per second
	EstimatedRemaining time.Duration
}

// StreamingLoader orchestrates the streaming loading process
type StreamingLoader struct {
	filename string
	world    *ecs.World
	config   StreamingConfig

	parseProgress  chan Progress
	entityProgress chan EntityProgress
```

```go
	totalStartTime time.Time
	loadingStats   LoadingStats
}

// StreamingConfig holds all streaming configuration
type StreamingConfig struct {
	ParseBatchSize   int
	ParseBufferSize  int
	MaxParseMemory   int64
	EntityBatchSize  int
	PreAllocateCount int
	MaxWorkers       int
	ProgressInterval time.Duration
	GCInterval       time.Duration
	MemoryLimit      int64
}

// LoadingStats holds comprehensive loading statistics
type LoadingStats struct {
	TotalEntities int64
	LoadingTime   time.Duration
	ParseRate     float64
	CreationRate  float64
	MemoryUsage   int64
	GCCount       int
	PulseRate     float64
}

// DefaultStreamingConfig returns optimized default configuration for large files
func DefaultStreamingConfig() StreamingConfig {
	return StreamingConfig{
		ParseBatchSize:   10000,
		ParseBufferSize:  4 * 1024 * 1024,
		MaxParseMemory:   1 * 1024 * 1024 * 1024,
		EntityBatchSize:  10000,
		PreAllocateCount: 500000,
		MaxWorkers:       runtime.NumCPU() * 2,
		ProgressInterval: 1 * time.Second,
		GCInterval:       5 * time.Second,
		MemoryLimit:      2 * 1024 * 1024 * 1024,
	}
}

// NewStreamingLoader creates a new streaming loader
func NewStreamingLoader(filename string, world *ecs.World, config StreamingConfig) *StreamingLoader {
	return &StreamingLoader{
		filename:       filename,
		world:          world,
		config:         config,
		parseProgress:  make(chan Progress, 10),
		entityProgress: make(chan EntityProgress, 10),
		totalStartTime: time.Now(),
	}
}

// Load performs the complete streaming load operation
func (sl *StreamingLoader) Load(ctx context.Context) (*LoadingStats, error) {
	fmt.Printf("Starting streaming load of %s...\n", sl.filename)

	var batchChan <-chan MonitorBatch
	var errorChan <-chan error

	parseConfig := ParseConfig{
		BatchSize:    sl.config.ParseBatchSize,
		BufferSize:   sl.config.ParseBufferSize,
		MaxMemory:    sl.config.MaxParseMemory,
		ProgressChan: sl.parseProgress,
	}

	if strings.HasSuffix(strings.ToLower(sl.filename), ".json") {
		jsonParser, err := NewStreamingJsonParser(sl.filename, parseConfig)
		if err != nil {
			return nil, fmt.Errorf("failed to create JSON parser: %w", err)
		}
		batchChan, errorChan = jsonParser.ParseBatches(ctx, sl.parseProgress)
	} else {
		yamlParser, err := NewStreamingYamlParser(sl.filename, parseConfig)
		if err != nil {
			return nil, fmt.Errorf("failed to create YAML parser: %w", err)
```

```
		}
		batchChan, errorChan = yamlParser.ParseBatches(ctx, sl.parseProgress)
	}

	entityCreator := NewStreamingEntityCreator(sl.world, EntityCreationConfig{
		BatchSize:    sl.config.EntityBatchSize,
		PreAllocate:  sl.config.PreAllocateCount,
		ProgressChan: sl.entityProgress,
	})

	err := entityCreator.ProcessBatches(ctx, batchChan, sl.entityProgress)
	if err != nil {
		return nil, fmt.Errorf("failed to create entities: %w", err)
	}

	select {
	case parseErr := <-errorChan:
		if parseErr != nil {
			return nil, fmt.Errorf("parsing error: %w", parseErr)
		}
	default:
	}

	sl.finalizeStats(entityCreator)

	fmt.Printf("Streaming load completed: %d entities in %v (%.0f entities/sec)\n",
		sl.loadingStats.TotalEntities,
		sl.loadingStats.LoadingTime,
		sl.loadingStats.CreationRate)

	return &sl.loadingStats, nil
}

func (sl *StreamingLoader) finalizeStats(creator *StreamingEntityCreator) {
	sl.loadingStats.LoadingTime = time.Since(sl.totalStartTime)
	entitiesCreated, _, creationRate := creator.GetStats()
	sl.loadingStats.TotalEntities = entitiesCreated
	sl.loadingStats.CreationRate = creationRate
	sl.loadingStats.PulseRate = creator.PulseRate()
	var m runtime.MemStats
	runtime.ReadMemStats(&m)
	sl.loadingStats.MemoryUsage = int64(m.Alloc)
}


/* loader/streaming/streaming_yaml_parser.go */

package streaming

import (
	"context"
	"fmt"
	"io"
	"os"

	"cpra/internal/loader/schema"
	"gopkg.in/yaml.v3"
)

// StreamingYamlParser handles true streaming parsing of a YAML file.
// It reads the file document by document, creating batches without loading the entire file into memory.
type StreamingYamlParser struct {
	filename string
	config   ParseConfig
}

// NewStreamingYamlParser creates a new streaming YAML parser.
func NewStreamingYamlParser(filename string, config ParseConfig) (*StreamingYamlParser, error) {
	return &StreamingYamlParser{
		filename: filename,
		config:   config,
	}, nil
}

// ParseBatches streams the YAML file and sends batches of monitors over a channel.
func (p *StreamingYamlParser) ParseBatches(ctx context.Context, progressChan chan<- Progress) (<-chan MonitorBatch, <-chan error) {
	batchChan := make(chan MonitorBatch, 100)
	errorChan := make(chan error, 1)
```

```go
	go func() {
		defer close(batchChan)
		defer close(errorChan)

		if err := p.parseFile(ctx, batchChan, progressChan); err != nil {
			errorChan <- err
		}
	}()

	return batchChan, errorChan
}

// parseFile performs the actual streaming YAML parsing.
// It decodes the main `monitors` list and then streams each monitor entry.
func (p *StreamingYamlParser) parseFile(ctx context.Context, batchChan chan<- MonitorBatch,
progressChan chan<- Progress) error {
	file, err := os.Open(p.filename)
	if err != nil {
		return fmt.Errorf("failed to open file: %w", err)
	}
	defer file.Close()

	decoder := yaml.NewDecoder(file)

	// The YAML file is expected to have a root structure like:
	// monitors:
	//   - name: ...
	//   - name: ...
	// We need to find the 'monitors' sequence node.

	// 1. Decode the top-level structure.
	var topLevel struct {
		Monitors yaml.Node `yaml:"monitors"`
	}
	if err := decoder.Decode(&topLevel); err != nil {
		if err == io.EOF {
			return nil // Empty file is not an error
		}
		return fmt.Errorf("failed to decode top-level 'monitors' field: %w", err)
	}

	// 2. Check if 'monitors' is a sequence.
	if topLevel.Monitors.Kind != yaml.SequenceNode {
		return fmt.Errorf("'monitors' field must be a YAML sequence")
	}

	// 3. Iterate through the sequence and decode each monitor.
	batchID := 0
	batch := make([]schema.Monitor, 0, p.config.BatchSize)

	for _, monitorNode := range topLevel.Monitors.Content {
		select {
		case <-ctx.Done():
			return ctx.Err()
		default:
			var monitor schema.Monitor
			if err := monitorNode.Decode(&monitor); err != nil {
				// Provide context for the decoding error.
				return fmt.Errorf("failed to decode monitor at line %d: %w",
monitorNode.Line, err)
			}

			// Basic validation to catch empty monitors from malformed YAML (e.g., "- ").
			if monitor.Name == "" && monitor.Pulse.Type == "" {
				// This is likely an empty or malformed monitor entry, so we skip it.
				continue
			}

			batch = append(batch, monitor)

			if len(batch) >= p.config.BatchSize {
				// Send the full batch.
				batchChan <- MonitorBatch{Monitors: batch, BatchID: batchID}
				// Reset batch.
				batch = make([]schema.Monitor, 0, p.config.BatchSize)
				batchID++
			}
		}
	}
```

```
	// 4. Send any remaining monitors in the last batch.
	if len(batch) > 0 {
		batchChan <- MonitorBatch{Monitors: batch, BatchID: batchID}
	}

	return nil
}

/* loader/validator/errors.go */

package validator


/* loader/validator/validator.go */

package validator

//
//import (
//	"cpra/internal/loader/parser"
//	parser2 "cpra/internal/loader/schema"
//)
//
//type Validator interface {
//	ValidateManifest() error
//}
//
//type YamlValidator struct {
//}
//
//func NewYamlValidator() *YamlValidator {
//	return &YamlValidator{}
//}
//
//func validateStringSize(s string, min int, max int) error {
//	if len(s) < min || len(s) > max {
//		return &parser.requiredFieldError{
//			Field:  "monitors.name",
//			Reason: "must be between 1 and 100 characters",
//		}
//	}
//	return nil
//}
//
//func (y *YamlValidator) validatePulseConfig(p parser2.PulseConfig) error {
//
//	switch p.(type) {
//	case parser2.PulseHTTPConfig:
//		if p.(parser2.PulseHTTPConfig).Url == "" {
//			return &parser.requiredFieldError{
//				Field:  "monitors.config.url",
//				Reason: "cannot be empty for pulse type http",
//			}
//		}
//	}
//
//	return nil
//}
//
//func (y *YamlValidator) validatePulse(p *parser2.Pulse) error {
//	if p.Type == "" {
//		return &parser.requiredFieldError{
//			Field:  "monitors.name",
//			Reason: "cannot be empty",
//		}
//	}
//
//	cfg, err := parser2.DecodePulseConfig(p)
//	if err != nil {
//		return err
//	}
//	if cfg == nil {
//		return &parser.requiredFieldError{
//			Field:  "monitors.pulse.config",
//			Reason: "cannot be empty",
//		}
//	}
//	err = y.validatePulseConfig(cfg)
//	if err != nil {
```

```go
//		return err
//	}
//
//	return nil
//}
//
//func (y *YamlValidator) validateMonitor(m *parser2.Monitor) error {
//	if m.Name == "" {
//		return &parser.requiredFieldError{
//			Field:  "Monitor.Name",
//			Reason: "cannot be empty",
//		}
//
//	}
//	return nil
//}
//
//func (y *YamlValidator) ValidateManifest(m *parser2.Manifest) error {
//	for _, monitor := range m.Monitors {
//		err := y.validateMonitor(&monitor)
//		if err != nil {
//			return err
//		}
//	}
//	return nil
//}


/* loader/validator/validator_test.go */

package validator


/* main.go */

package main

import (
	"context"
	"flag"
	"fmt"
	"log"
	"os"
	"os/signal"
	"runtime"
	"runtime/pprof"
	"sync"
	"syscall"
	"time"

	"cpra/internal/controller"
)

func main() {
	// Command line flags
	var (
		configFile = flag.String("config", "", "Configuration file path")
		yamlFile   = flag.String("yaml", "internal/loader/replicated_test.yaml", "YAML file 
with monitors")
		profile    = flag.Bool("profile", false, "Enable CPU profiling")
		debug      = flag.Bool("debug", false, "Enable debug logging")
	)
	flag.Parse()

	// Initialize loggers first
	controller.InitializeLoggers(*debug)

	controller.SystemLogger.Info("Starting CPRA Optimized Controller for 1M Monitors")
	controller.SystemLogger.Info("Input file: %s", *yamlFile)

	// Create optimized configuration
	config := controller.DefaultConfig()

	// Override configuration if file provided
	if *configFile != "" {
		fmt.Printf("Loading configuration from: %s\n", *configFile)
		// Configuration loading would be implemented here
	}

	// Enable profiling if requested
```

```go
	if *profile {
		fmt.Println("CPU profiling enabled")
		f, err := os.Create("cpu.pprof")
		if err != nil {
			log.Fatal("could not create CPU profile: ", err)
		}
		defer f.Close() // error handling omitted for example
		if err := pprof.StartCPUProfile(f); err != nil {
			log.Fatal("could not start CPU profile: ", err)
		}
		defer pprof.StopCPUProfile()
	}

	// Create the new optimized controller
	oc := controller.NewOptimizedController(config)

	// Setup context for graceful shutdown
	ctx, cancel := context.WithCancel(context.Background())
	defer cancel()

	// Setup signal handler for graceful shutdown
	sigChan := make(chan os.Signal, 1)
	signal.Notify(sigChan, syscall.SIGINT, syscall.SIGTERM)

	var shutdownInitiated bool
	var shutdownMutex sync.Mutex

	go func() {
		sig := <-sigChan
		shutdownMutex.Lock()
		if !shutdownInitiated {
			shutdownInitiated = true
			fmt.Printf("\nShutdown signal received (%v)...\n", sig)
			cancel()
		}
		shutdownMutex.Unlock()
	}()

	// Load monitors if YAML file exists
	if _, err := os.Stat(*yamlFile); err == nil {
		fmt.Printf("Loading monitors from %s...\n", *yamlFile)
		start := time.Now()

		if err := oc.LoadMonitors(ctx, *yamlFile); err != nil {
			fmt.Printf("Error loading monitors: %v\n", err)
			os.Exit(1)
		}

		fmt.Printf("Monitor loading completed in %v\n", time.Since(start))
	} else {
		fmt.Printf("Warning: YAML file %s not found, starting without loading monitors\n",
*yamlFile)
	}

	// Start the optimized controller
	if err := oc.Start(ctx); err != nil {
		fmt.Printf("Error starting controller: %v\n", err)
		os.Exit(1)
	}

	// Wait for shutdown signal
	<-ctx.Done()
	fmt.Println("Shutting down...")

	// Print memory Usage
	PrintMemUsage()

	// Stop the controller
	oc.Stop()

	// Close loggers after everything is done
	controller.CloseLoggers()

	fmt.Println("CPRA Optimized Controller stopped")
}

// bToMb converts bytes to megabytes
func bToMb(b uint64) uint64 {
	return b / 1024 / 1024
}
```

```go
// PrintMemUsage outputs the current, total, and system memory usage
func PrintMemUsage() {
	var m runtime.MemStats
	runtime.ReadMemStats(&m)
	fmt.Printf("\nMemory usage on exit:\n")
	fmt.Printf("Alloc = %v MiB", bToMb(m.Alloc))
	fmt.Printf("\tTotalAlloc = %v MiB", bToMb(m.TotalAlloc))
	fmt.Printf("\tSys = %v MiB", bToMb(m.Sys))
	fmt.Printf("\tNumGC = %v\n", m.NumGC)
}


/* queue/adaptive_queue.go */

package queue

import (
	"cpra/internal/jobs"
	"errors"
	"reflect"
	"sync/atomic"
	"time"
)

// AdaptiveQueue is a lock-free, thread-safe, fixed-size circular queue.
// It is designed for high-throughput scenarios with multiple producers and consumers.
// It implements the Queue interface.
type AdaptiveQueue struct {
	buffer   []jobs.Job
	capacity uint64
	head     uint64
	tail     uint64
	closed   int32

	enqueuedCount       int64
	dequeuedCount       int64
	totalQueueWaitNanos int64
	maxQueueWaitNanos   int64
	startUnixNano       int64
	lastEnqueueUnixNano int64
	lastDequeueUnixNano int64
}

// NewAdaptiveQueue creates a new AdaptiveQueue with the given capacity.
// Capacity must be a power of 2 for efficient bitwise operations.
func NewAdaptiveQueue(capacity uint64) (*AdaptiveQueue, error) {
	if (capacity & (capacity - 1)) != 0 {
		return nil, errors.New("capacity must be a power of 2")
	}
	queue := &AdaptiveQueue{
		buffer:        make([]jobs.Job, capacity),
		capacity:      capacity,
		startUnixNano: time.Now().UnixNano(),
	}
	return queue, nil
}

// Enqueue adds a single job to the queue.
func (q *AdaptiveQueue) Enqueue(job jobs.Job) error {
	if atomic.LoadInt32(&q.closed) == 1 {
		return ErrQueueClosed
	}

	now := time.Now()
	for {
		head := atomic.LoadUint64(&q.head)
		tail := atomic.LoadUint64(&q.tail)

		if tail-head >= q.capacity {
			return ErrQueueFull // Queue is full
		}

		// Attempt to claim the next spot
		if atomic.CompareAndSwapUint64(&q.tail, tail, tail+1) {
			if !isNilJob(job) {
				job.SetEnqueueTime(now)
			}
			q.buffer[tail&(q.capacity-1)] = job
			atomic.AddInt64(&q.enqueuedCount, 1)
```

```go
				atomic.StoreInt64(&q.lastEnqueueUnixNano, now.UnixNano())
				return nil
			}
		}
}

// EnqueueBatch adds a batch of jobs to the queue using a highly concurrent, lock-free algorithm.
func (q *AdaptiveQueue) EnqueueBatch(jobsInterface []interface{}) error {
	if len(jobsInterface) == 0 {
		return nil
	}

	// Convert interface{} slice to jobs.Job slice
	convertedJobs := make([]jobs.Job, len(jobsInterface))
	for i, job := range jobsInterface {
		if j, ok := job.(jobs.Job); ok {
			convertedJobs[i] = j
		} else {
			return errors.New("invalid job type in batch")
		}
	}
	if atomic.LoadInt32(&q.closed) == 1 {
		return ErrQueueClosed
	}
	n := uint64(len(convertedJobs))

	now := time.Now()
	for {
		head := atomic.LoadUint64(&q.head)
		tail := atomic.LoadUint64(&q.tail)

		if tail-head+n > q.capacity {
			return ErrQueueFull
		}

		// Atomically claim a slot for the entire batch
		if atomic.CompareAndSwapUint64(&q.tail, tail, tail+n) {
			// Once the slot is claimed, we can write the batch without further atomics
			for i := uint64(0); i < n; i++ {
				job := convertedJobs[i]
				if !isNilJob(job) {
					job.SetEnqueueTime(now)
				}
				q.buffer[(tail+i)&(q.capacity-1)] = job
			}
			atomic.AddInt64(&q.enqueuedCount, int64(n))
			atomic.StoreInt64(&q.lastEnqueueUnixNano, now.UnixNano())
			return nil
		}
		// If CAS fails, another producer got there first. Loop and try again.
	}
}

// DequeueBatch removes and returns a batch of jobs from the queue.
func (q *AdaptiveQueue) DequeueBatch(maxSize int) ([]jobs.Job, error) {
	if atomic.LoadInt32(&q.closed) == 1 && q.IsEmpty() {
		return nil, ErrQueueClosed
	}

	for {
		head := atomic.LoadUint64(&q.head)
		tail := atomic.LoadUint64(&q.tail)

		if head >= tail {
			return nil, nil // Queue is empty
		}

		n := tail - head
		if n > uint64(maxSize) {
			n = uint64(maxSize)
		}

		// Atomically claim the batch for dequeuing
		if atomic.CompareAndSwapUint64(&q.head, head, head+n) {
			batch := make([]jobs.Job, n)
			now := time.Now()
			for i := uint64(0); i < n; i++ {
				batch[i] = q.buffer[(head+i)&(q.capacity-1)]
				// Nil out the buffer slot to help the GC
				q.buffer[(head+i)&(q.capacity-1)] = nil
```

```
				if !isNilJob(batch[i]) {
					enqueueTime := batch[i].GetEnqueueTime()
					if !enqueueTime.IsZero() {
						wait := now.Sub(enqueueTime)
						atomic.AddInt64(&q.totalQueueWaitNanos, int64(wait))
						for {
							currentMax :=
atomic.LoadInt64(&q.maxQueueWaitNanos)
							if waitNs := int64(wait); waitNs <= currentMax
{
								break
							}
							if
atomic.CompareAndSwapInt64(&q.maxQueueWaitNanos, currentMax, int64(wait)) {
								break
							}
						}
					}
				}
			}
			atomic.AddInt64(&q.dequeuedCount, int64(n))
			atomic.StoreInt64(&q.lastDequeueUnixNano, now.UnixNano())
			return batch, nil
		}
		// If CAS fails, another consumer got there first. Loop and try again.
	}
}

// Dequeue removes and returns a single job from the queue.
func (q *AdaptiveQueue) Dequeue() (jobs.Job, error) {
	if atomic.LoadInt32(&q.closed) == 1 && q.IsEmpty() {
		return nil, ErrQueueClosed
	}

	for {
		head := atomic.LoadUint64(&q.head)
		tail := atomic.LoadUint64(&q.tail)

		if head >= tail {
			return nil, nil // Queue is empty
		}

		job := q.buffer[head&(q.capacity-1)]

		// Attempt to move the head pointer
		if atomic.CompareAndSwapUint64(&q.head, head, head+1) {
			now := time.Now()
			if !isNilJob(job) {
				enqueueTime := job.GetEnqueueTime()
				if !enqueueTime.IsZero() {
					wait := now.Sub(enqueueTime)
					atomic.AddInt64(&q.totalQueueWaitNanos, int64(wait))
					for {
						currentMax := atomic.LoadInt64(&q.maxQueueWaitNanos)
						if waitNs := int64(wait); waitNs <= currentMax {
							break
						}
						if atomic.CompareAndSwapInt64(&q.maxQueueWaitNanos,
currentMax, int64(wait)) {
							break
						}
					}
				}
			}
			atomic.AddInt64(&q.dequeuedCount, 1)
			atomic.StoreInt64(&q.lastDequeueUnixNano, now.UnixNano())
			return job, nil
		}
	}
}

// IsEmpty checks if the queue is empty.
func (q *AdaptiveQueue) IsEmpty() bool {
	return atomic.LoadUint64(&q.head) == atomic.LoadUint64(&q.tail)
}

// Close marks the queue as closed.
func (q *AdaptiveQueue) Close() {
	atomic.StoreInt32(&q.closed, 1)
```

```go
}

func isNilJob(job jobs.Job) bool {
	if job == nil {
		return true
	}
	val := reflect.ValueOf(job)
	switch val.Kind() {
	case reflect.Ptr, reflect.Interface, reflect.Slice, reflect.Map, reflect.Func, reflect.Chan:
		return val.IsNil()
	default:
		return false
	}
}

// Stats returns the current statistics for the queue.
// Note: This is a simplified version. A full implementation would track more metrics.
func (q *AdaptiveQueue) Stats() Stats {
	head := atomic.LoadUint64(&q.head)
	tail := atomic.LoadUint64(&q.tail)
	depth := tail - head
	enq := atomic.LoadInt64(&q.enqueuedCount)
	deq := atomic.LoadInt64(&q.dequeuedCount)
	elapsed := time.Since(time.Unix(0, atomic.LoadInt64(&q.startUnixNano)))
	if elapsed <= 0 {
		elapsed = time.Millisecond
	}
	avgWaitNs := int64(0)
	if deq > 0 {
		avgWaitNs = atomic.LoadInt64(&q.totalQueueWaitNanos) / deq
	}
	stats := Stats{
		QueueDepth:   int(depth),
		Capacity:     int(q.capacity),
		Enqueued:     enq,
		Dequeued:     deq,
		Dropped:      0,
		MaxQueueTime: time.Duration(atomic.LoadInt64(&q.maxQueueWaitNanos)),
		AvgQueueTime: time.Duration(avgWaitNs),
		EnqueueRate:  float64(enq) / elapsed.Seconds(),
		DequeueRate:  float64(deq) / elapsed.Seconds(),
		LastEnqueue:  time.Unix(0, atomic.LoadInt64(&q.lastEnqueueUnixNano)),
		LastDequeue:  time.Unix(0, atomic.LoadInt64(&q.lastDequeueUnixNano)),
		SampleWindow: elapsed,
	}
	return stats
}


/* queue/bounded_queue.go */

package queue

import (
	"errors"
	"sync"
	"sync/atomic"
	"time"

	"cpra/internal/jobs"
)

var (
	ErrQueueFull   = errors.New("queue is full")
	ErrQueueClosed = errors.New("queue is closed")
)

// BoundedQueue implements a high-performance bounded queue with batching.
// It now implements the queue.Queue interface.
type BoundedQueue struct {
	batches      chan []jobs.Job
	maxSize      int32
	maxBatch     int32
	closed       int32
	enqueued     int64
	dequeued     int64
	dropped      int64
	mu           sync.RWMutex
	batchTimeout time.Duration
}
```

```go
// BoundedQueueConfig holds queue configuration.
type BoundedQueueConfig struct {
	MaxSize      int
	MaxBatch     int
	BatchTimeout time.Duration
}

// NewBoundedQueue creates a new bounded queue.
func NewBoundedQueue(config BoundedQueueConfig) *BoundedQueue {
	return &BoundedQueue{
		batches:      make(chan []jobs.Job, config.MaxSize),
		maxSize:      int32(config.MaxSize),
		maxBatch:     int32(config.MaxBatch),
		batchTimeout: config.BatchTimeout,
	}
}

// Enqueue adds a single job to the queue.
func (q *BoundedQueue) Enqueue(job jobs.Job) error {
	return q.EnqueueBatch([]jobs.Job{job})
}

// EnqueueBatch adds a batch of jobs to the queue.
func (q *BoundedQueue) EnqueueBatch(jobs []jobs.Job) error {
	if atomic.LoadInt32(&q.closed) == 1 {
		return ErrQueueClosed
	}

	if len(jobs) == 0 {
		return nil
	}

	enqueueTime := time.Now()
	for _, job := range jobs {
		job.SetEnqueueTime(enqueueTime)
	}

	batch := jobs
	if len(batch) > int(q.maxBatch) {
		batch = batch[:q.maxBatch]
	}

	select {
	case q.batches <- batch:
		atomic.AddInt64(&q.enqueued, int64(len(batch)))
		return nil
	default:
		atomic.AddInt64(&q.dropped, int64(len(batch)))
		return ErrQueueFull
	}
}

// Dequeue removes and returns a single job from the queue.
func (q *BoundedQueue) Dequeue() (jobs.Job, error) {
	// This is inefficient for a bounded queue, but it satisfies the interface.
	// The adaptive queue will have a proper single-item dequeue.
	select {
	case batch, ok := <-q.batches:
		if !ok {
			return nil, ErrQueueClosed
		}
		atomic.AddInt64(&q.dequeued, int64(len(batch)))
		if len(batch) > 1 {
			// Re-enqueue the rest of the batch. This is very inefficient.
			go func() { q.batches <- batch[1:] }()
		}
		return batch[0], nil
	case <-time.After(10 * time.Millisecond): // Non-blocking with a small timeout
		return nil, nil // Queue is empty
	}
}

// DequeueBatch removes a batch of jobs from the queue.
func (q *BoundedQueue) DequeueBatch(maxSize int) ([]jobs.Job, error) {
	select {
	case batch, ok := <-q.batches:
		if !ok {
			return nil, ErrQueueClosed
		}
```

```go
		atomic.AddInt64(&q.dequeued, int64(len(batch)))
		if len(batch) > maxSize {
			// This is inefficient, but necessary to respect maxSize.
			go func() { q.batches <- batch[maxSize:] }()
			return batch[:maxSize], nil
		}
		return batch, nil
	default:
		// Non-blocking, return empty if no batch is immediately available.
		return nil, nil
	}
}

// Close closes the queue.
func (q *BoundedQueue) Close() {
	if atomic.CompareAndSwapInt32(&q.closed, 0, 1) {
		close(q.batches)
	}
}

// Stats returns current queue statistics.
func (q *BoundedQueue) Stats() Stats {
	return Stats{
		Enqueued:   atomic.LoadInt64(&q.enqueued),
		Dequeued:   atomic.LoadInt64(&q.dequeued),
		Dropped:    atomic.LoadInt64(&q.dropped),
		QueueDepth: len(q.batches),
		Capacity:   int(q.maxSize),
	}
}


/* queue/dynamic_worker_pool.go */

package queue

import (
	"context"
	"cpra/internal/jobs"
	"log"
	"math"
	"sync"
	"sync/atomic"
	"time"

	"github.com/panjf2000/ants/v2"
)

// ResultRouter handles routing of job results to type-specific channels.
// This enables decoupling result processing from the main worker pool.
type ResultRouter struct {
	PulseResultChan        chan []jobs.Result
	InterventionResultChan chan []jobs.Result
	CodeResultChan         chan []jobs.Result

	config WorkerPoolConfig
	logger *log.Logger
}

// WorkerPoolStats exposes runtime metrics for the dynamic worker pool.
type WorkerPoolStats struct {
	MinWorkers      int
	MaxWorkers      int
	CurrentCapacity int
	RunningWorkers  int
	WaitingTasks    int
	TargetWorkers   int
	TasksSubmitted  int64
	TasksCompleted  int64
	ScalingEvents   int64
	LastScaleTime   time.Time
	PendingResults  int
}

// NewResultRouter creates a new result router with buffered channels.
func NewResultRouter(config WorkerPoolConfig, logger *log.Logger) *ResultRouter {
	bufferSize := config.MaxWorkers // Buffer size based on max workers
	return &ResultRouter{
		PulseResultChan:        make(chan []jobs.Result, bufferSize),
		InterventionResultChan: make(chan []jobs.Result, bufferSize),
```

```
		CodeResultChan:         make(chan []jobs.Result, bufferSize),
		config:                 config,
		logger:                 logger,
	}
}

// RouteResults takes a batch of mixed results and routes them to appropriate channels.
func (r *ResultRouter) RouteResults(results []jobs.Result) {
	if len(results) == 0 {
		return
	}

	// Group results by type
	pulseResults := make([]jobs.Result, 0, len(results))
	interventionResults := make([]jobs.Result, 0, len(results))
	codeResults := make([]jobs.Result, 0, len(results))

	for _, result := range results {
		switch result.Payload["type"] {
		case "pulse":
			pulseResults = append(pulseResults, result)
		case "intervention":
			interventionResults = append(interventionResults, result)
		case "code":
			codeResults = append(codeResults, result)
		default:
			r.logger.Printf("Unknown job type in result: %v", result.Payload["type"])
		}
	}

	// Send to appropriate channels with backpressure logging
	if len(pulseResults) > 0 {
		r.sendWithBackpressure(r.PulseResultChan, pulseResults, "pulse")
	}
	if len(interventionResults) > 0 {
		r.sendWithBackpressure(r.InterventionResultChan, interventionResults, "intervention")
	}
	if len(codeResults) > 0 {
		r.sendWithBackpressure(r.CodeResultChan, codeResults, "code")
	}
}

func (r *ResultRouter) sendWithBackpressure(ch chan []jobs.Result, batch []jobs.Result, label string)
{
	backoff := r.config.ResultBatchTimeout
	if backoff <= 0 {
		backoff = 50 * time.Millisecond
	}
	ticker := time.NewTicker(backoff)
	defer ticker.Stop()

	for {
		select {
		case ch <- batch:
			return
		case <-ticker.C:
			r.logger.Printf("Backpressure: %s results stalled (%d jobs waiting)", label,
len(batch))
		}
	}
}

// Close closes all result channels.
func (r *ResultRouter) Close() {
	close(r.PulseResultChan)
	close(r.InterventionResultChan)
	close(r.CodeResultChan)
}

// DynamicWorkerPool manages a pool of workers that execute jobs from a queue.
// It can dynamically adjust the number of workers based on load.
type DynamicWorkerPool struct {
	queue      Queue
	antsPool   *ants.PoolWithFunc
	logger     *log.Logger
	config     WorkerPoolConfig
	resultChan chan jobs.Result
	router     *ResultRouter

	ctx    context.Context
```

```go
	cancel context.CancelFunc
	wg     sync.WaitGroup

	tasksSubmitted int64
	tasksCompleted int64
	scalingEvents  int64
	lastTarget     int64
	lastScaleTime  int64
	stopping       int32
}

// WorkerPoolConfig holds configuration for the DynamicWorkerPool.
type WorkerPoolConfig struct {
	MinWorkers         int
	MaxWorkers         int
	AdjustmentInterval time.Duration
	ResultBatchSize    int
	ResultBatchTimeout time.Duration
	TargetQueueLatency time.Duration
}

// DefaultWorkerPoolConfig returns a default configuration for the worker pool.
func DefaultWorkerPoolConfig() WorkerPoolConfig {
	return WorkerPoolConfig{
		MinWorkers:         10,
		MaxWorkers:         10000,
		AdjustmentInterval: 5 * time.Second,
		ResultBatchSize:    1000,
		ResultBatchTimeout: 10 * time.Millisecond,
		TargetQueueLatency: 100 * time.Millisecond,
	}
}

// NewDynamicWorkerPool creates a new dynamic worker pool.
func NewDynamicWorkerPool(q Queue, config WorkerPoolConfig, logger *log.Logger) (*DynamicWorkerPool,
error) {
	if config.MinWorkers <= 0 {
		config.MinWorkers = 1
	}
	if config.MaxWorkers < config.MinWorkers {
		config.MaxWorkers = config.MinWorkers
	}
	if config.ResultBatchSize <= 0 {
		config.ResultBatchSize = config.MaxWorkers
	}
	if config.TargetQueueLatency <= 0 {
		config.TargetQueueLatency = 100 * time.Millisecond
	}

	ctx, cancel := context.WithCancel(context.Background())

	pool := &DynamicWorkerPool{
		queue:      q,
		logger:     logger,
		config:     config,
		resultChan: make(chan jobs.Result, config.MaxWorkers),
		router:     NewResultRouter(config, logger),
		ctx:        ctx,
		cancel:     cancel,
	}

	workerFunc := func(job interface{}) {
		j, ok := job.(jobs.Job)
		if !ok {
			pool.logger.Printf("Error: Invalid job type in worker pool: %T", job)
			return
		}
		result := j.Execute()
		if atomic.LoadInt32(&pool.stopping) == 1 {
			return
		}
		select {
		case pool.resultChan <- result:
		case <-pool.ctx.Done():
		}
	}

	antsPool, err := ants.NewPoolWithFunc(config.MaxWorkers, workerFunc,
ants.WithPanicHandler(func(err interface{}) {
		pool.logger.Printf("Worker panic: %v", err)
```

```go
	}))
	if err != nil {
		return nil, err
	}
	pool.antsPool = antsPool
	pool.antsPool.Tune(config.MinWorkers)
	atomic.StoreInt64(&pool.lastTarget, int64(config.MinWorkers))
	atomic.StoreInt64(&pool.lastScaleTime, time.Now().UnixNano())

	return pool, nil
}

// Start begins the worker pool's operations.
func (p *DynamicWorkerPool) Start() {
	routineCount := 2
	if p.config.AdjustmentInterval > 0 {
		routineCount++
	}
	p.wg.Add(routineCount)
	go p.dispatcher()
	go p.resultProcessor()
	if p.config.AdjustmentInterval > 0 {
		go p.autoScale()
	}
	p.logger.Println("DynamicWorkerPool started")
}

// GetRouter returns the result router for accessing type-specific result channels.
func (p *DynamicWorkerPool) GetRouter() *ResultRouter {
	return p.router
}

// DrainAndStop waits for outstanding tasks to finish before stopping the worker pool.
func (p *DynamicWorkerPool) DrainAndStop() {
	if !atomic.CompareAndSwapInt32(&p.stopping, 0, 1) {
		return
	}
	p.logger.Println("Draining DynamicWorkerPool...")
	p.cancel()
	done := make(chan struct{})
	go func() {
		p.wg.Wait()
		close(done)
	}()
	select {
	case <-done:
	case <-time.After(p.config.TargetQueueLatency * 5):
		p.logger.Println("Draining timed out, continuing shutdown")
	}
	remaining := len(p.resultChan)
	if remaining > 0 {
		p.logger.Printf("Flushing %d queued results before close", remaining)
	}
	close(p.resultChan)
	p.router.Close()
	p.antsPool.Release()
	p.logger.Println("DynamicWorkerPool stopped")
}

// dispatcher fetches batches of jobs from the queue and submits them to the ants pool.
func (p *DynamicWorkerPool) dispatcher() {
	defer p.wg.Done()
	for {
		select {
		case <-p.ctx.Done():
			return
		default:
			batchTarget := p.antsPool.Cap()
			if batchTarget <= 0 {
				batchTarget = p.config.MinWorkers
			}
			if batchTarget > p.config.ResultBatchSize {
				batchTarget = p.config.ResultBatchSize
			}
			if batchTarget <= 0 {
				batchTarget = 1
			}

			jobs, err := p.queue.DequeueBatch(batchTarget)
			if err != nil {
```

```go
				if err != ErrQueueClosed {
					p.logger.Printf("Error dequeuing job batch: %v", err)
				}
				time.Sleep(100 * time.Millisecond) // Wait a bit if there's an error
				continue
			}
			if len(jobs) == 0 {
				time.Sleep(10 * time.Millisecond) // Wait if the queue is empty
				continue
			}

			atomic.AddInt64(&p.tasksSubmitted, int64(len(jobs)))

			for _, job := range jobs {
				if err := p.antsPool.Invoke(job); err != nil {
					p.logger.Printf("Error invoking job: %v", err)
				}
			}
		}
	}
}

// resultProcessor collects individual results and routes them through the router in batches.
func (p *DynamicWorkerPool) resultProcessor() {
	defer p.wg.Done()

	batch := make([]jobs.Result, 0, p.config.ResultBatchSize)
	ticker := time.NewTicker(p.config.ResultBatchTimeout)
	defer ticker.Stop()

	for {
		select {
		case <-p.ctx.Done():
			// Route any remaining results before shutting down
			if len(batch) > 0 {
				p.router.RouteResults(batch)
			}
			return
		case result, ok := <-p.resultChan:
			if !ok { // resultChan was closed
				if len(batch) > 0 {
					p.router.RouteResults(batch)
				}
				return
			}
			atomic.AddInt64(&p.tasksCompleted, 1)
			batch = append(batch, result)
			if len(batch) >= p.config.ResultBatchSize {
				p.router.RouteResults(batch)
				batch = make([]jobs.Result, 0, p.config.ResultBatchSize)
				// Reset the ticker to prevent immediate firing
				ticker.Reset(p.config.ResultBatchTimeout)
			}
		case <-ticker.C:
			// Route partial batches on timeout
			if len(batch) > 0 {
				p.router.RouteResults(batch)
				batch = make([]jobs.Result, 0, p.config.ResultBatchSize)
			}
		}
	}
}

// autoScale periodically tunes the ants pool capacity based on queue depth.
func (p *DynamicWorkerPool) autoScale() {
	defer p.wg.Done()

	ticker := time.NewTicker(p.config.AdjustmentInterval)
	defer ticker.Stop()

	for {
		select {
		case <-p.ctx.Done():
			return
		case <-ticker.C:
			stats := p.queue.Stats()
			desired := p.desiredCapacity(stats)
			current := p.antsPool.Cap()
			if desired != current {
				p.antsPool.Tune(desired)
```

```
                                p.logger.Printf("Tuned worker pool capacity to %d (queue depth=%d)",
desired, stats.QueueDepth)
                                atomic.StoreInt64(&p.lastTarget, int64(desired))
                                atomic.StoreInt64(&p.lastScaleTime, time.Now().UnixNano())
                                atomic.AddInt64(&p.scalingEvents, 1)
                        }
                }
        }
}

func (p *DynamicWorkerPool) desiredCapacity(stats QueueStats) int {
        current := p.antsPool.Cap()
        if current <= 0 {
                current = p.config.MinWorkers
        }

        minWorkers := p.config.MinWorkers
        maxWorkers := p.config.MaxWorkers
        if maxWorkers < minWorkers {
                maxWorkers = minWorkers
        }

        enqueueRate := stats.EnqueueRate
        if enqueueRate <= 0 {
                enqueueRate = stats.DequeueRate
        }
        targetLatency := p.config.TargetQueueLatency
        if targetLatency <= 0 {
                targetLatency = 100 * time.Millisecond
        }

        desired := current

        // Estimate per-worker throughput
        perWorker := 0.0
        if current > 0 && stats.DequeueRate > 0 {
                perWorker = stats.DequeueRate / float64(current)
        }
        if perWorker > 0 && enqueueRate > 0 {
                desired = int(math.Ceil(enqueueRate / perWorker))
        }

        // Enforce latency budget using Little's Law (L = λW)
        if enqueueRate > 0 {
                targetDepth := enqueueRate * targetLatency.Seconds()
                // Always allow at least minWorkers entities worth of backlog
                if targetDepth < float64(minWorkers) {
                        targetDepth = float64(minWorkers)
                }
                depth := float64(stats.QueueDepth)
                if depth > targetDepth && targetDepth > 0 {
                        scale := depth / targetDepth
                        desired = int(math.Ceil(float64(desired) * scale))
                } else if depth < targetDepth/2 && desired > minWorkers {
                        desired = int(math.Max(float64(minWorkers), math.Ceil(float64(desired)*0.8)))
                }
        }

        if desired < minWorkers {
                desired = minWorkers
        }
        if desired > maxWorkers {
                desired = maxWorkers
        }
        return desired
}

// Stats returns runtime statistics for the worker pool.
func (p *DynamicWorkerPool) Stats() WorkerPoolStats {
        return WorkerPoolStats{
                MinWorkers:      p.config.MinWorkers,
                MaxWorkers:      p.config.MaxWorkers,
                CurrentCapacity: p.antsPool.Cap(),
                RunningWorkers:  p.antsPool.Running(),
                WaitingTasks:    p.antsPool.Waiting(),
                TargetWorkers:   int(atomic.LoadInt64(&p.lastTarget)),
                TasksSubmitted:  atomic.LoadInt64(&p.tasksSubmitted),
                TasksCompleted:  atomic.LoadInt64(&p.tasksCompleted),
                ScalingEvents:   atomic.LoadInt64(&p.scalingEvents),
                LastScaleTime:   time.Unix(0, atomic.LoadInt64(&p.lastScaleTime)),
```

```go
		PendingResults:  len(p.resultChan),
	}
}


/* queue/queue.go */

package queue

import (
	"cpra/internal/jobs"
	"time"
)

// Queue defines the interface for a generic, thread-safe queue system.
// This allows the controller and systems to be decoupled from a specific queue implementation.
type Queue interface {
	// Enqueue adds a single job to the queue.
	Enqueue(job jobs.Job) error

	// EnqueueBatch adds a slice of jobs to the queue.
	EnqueueBatch(jobs []interface{}) error

	// Dequeue removes and returns a single job from the queue.
	Dequeue() (jobs.Job, error)

	// DequeueBatch removes and returns a batch of jobs from the queue.
	DequeueBatch(maxSize int) ([]jobs.Job, error)

	// Close shuts down the queue and prevents new jobs from being enqueued.
	Close()

	// Stats returns statistics about the queue's performance.
	Stats() Stats
}

// Stats holds performance metrics for a queue.
type Stats struct {
	QueueDepth    int
	Capacity      int
	Enqueued      int64
	Dequeued      int64
	Dropped       int64
	MaxQueueTime  time.Duration
	AvgQueueTime  time.Duration
	MaxJobLatency time.Duration
	AvgJobLatency time.Duration
	EnqueueRate   float64 // jobs per second since queue creation
	DequeueRate   float64 // jobs per second since queue creation
	LastEnqueue   time.Time
	LastDequeue   time.Time
	SampleWindow  time.Duration // elapsed time used for rate calculations
}
```